RUNNING DEEP NEURAL NETWORKS USING DIVIDE AND CONQUER CHECKPOINTING AND TENSOR STREAMING

by

Hamad Ahmed

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

School of Electrical and Computer Engineering

West Lafayette, Indiana

December 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL STATEMENT OF COMMITTEE APPROVAL

Dr. Dimitrios Peroulis, Chair

School of Electrical and Computer Engineering

Dr. Kaushik Roy

School of Electrical and Computer Engineering

Dr. Jan P. Allebach

School of Electrical and Computer Engineering

Dr. Ananth Grama

Department of Computer Science

Dr. Suresh Jagannathan

Department of Computer Science

Approved by:

Dr. Dimitrios Peroulis

This thesis is dedicated to my family, friends and advisor for their unconditional support and help during this PhD.

ACKNOWLEDGMENTS

This work was supported, in part, by the US National Science Foundation under Grants 1522954-IIS and 1734938-IIS, by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior/Interior Business Center (DOI/IBC) contract number D17PC00341, and by Siemens Corporation, Corporate Technology. Any opinions, findings, views, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, official policies, or endorsements, either expressed or implied, of the sponsors. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation herein.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

m	meters
s	seconds
Hz	Hertz

ABBREVIATIONS

AI.	Artificial Intelligence
AD	Automatic Differentiation
GPU	Graphical Processing Unit
CPU	Central Processing Unit
MB	Megabyte
GB	Gigabyte
TB	Terabyte
RAM	Random Access Memory
DL	Deep Learning
GPT	Generative Pre-Training
CV	Computer Vision
CNN	Convolutional Neural Networks
VGG	Visual Geometry Group
GTX	Giga Texel shader eXtreme
BERT	Bidirectional Encoder Representations from Transformers
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
RTX	Ray-tracing Texel eXtreme
POPL	Principles of Programming Languages
ICFP	International Conference on Functional Programming
PLDI	Programming Languages Design & Implementation
CUDA	Compute Unified Device Architecture
cuDNN	NVIDIA CUDA Deep Neural Network library
COCO	Common Objects in Context
V100	NVIDIA Titan V100
MIMD	Multiple Instruction Multiple Data
SIMD	Single Instruction Multiple Data
2D	Two Dimensional
3D	Three Dimensional

PCIe	Peripheral Component Interconnect express
CSP	Constraint Satisfaction Problem
FLOP	Floating Point Operation
GC	Garbage Collection
vDNN	Virtualized Deep Neural Networks
moDNN	Memory Optimal Deep Neural Networks
ADAM	Adaptive Moment Estimation
AdaGrad	Adaptive Gradient Estimation
RMSProp	Root Mean Square Propagation
YOLO	You Only Look Once
EEG	Electroencephalography
MEVA	Multiview Extended Video Activities
TSM	Temporal Segment Networks
SVM	Support Vector Machines

ABSTRACT

The application of reverse mode automatic differentiation (AD) to a differentiable program requires saving the intermediate outputs of each operation on a data structure called the tape for use during the reverse sweep. This puts a bound on the length of the differentiable program or the depth of a deep neural network because memory is always limited and the tape has to be able to fit on the available memory. This problem is more severe for programs that run on the GPU because GPU memory is extremely limited (only 12GB for most consumer GPUs). Further, for parameterized programs like neural networks, where our goal is to compute the gradients of model parameters with respect to a scalar loss value, we also need to store these model parameters on the limited available memory. These model parameters can grow to be hundreds of GBs in size, inhibiting even the instantiation of such deep networks on the GPU. In this thesis research, I present Scorch, a new deep learning framework with built-in support for two key features: (1) divide-and-conquer checkpointing i.e. rearranging the application of reverse-mode AD to trade-off space vs. time by storing only parts of the tape at a certain time at the cost of recomputing the other parts later and (2) tensor streaming between the CPU RAM and GPU RAM synchronized with the execution of reverse-mode AD. Divide-and-conquer checkpointing lifts the memory bound caused by the tape as we can run a program that would typically have a huge tape on limited memory by only keeping a small portion of the tape live at any given time. Tensor streaming lifts the memory bound caused by the size of the model parameters as we use CPU RAM to store these parameters and stream them seamlessly to the GPU as required. These techniques allow us to run gradient descent on a long running differentiable program or a deep neural network of any arbitrary size. Scorch is evaluated on several large real-world examples and show that Scorch is able to create and run gradient descent on the popular image classification network ResNet with a depth of upto 250,000 layers, a popular image segmentation network DraNet with a depth of upto 250,000 layers and a popular language generation model GPT-3 with 175 billion parameters, all while maintaining highly efficient use of CPU and GPU resources.

1. INTRODUCTION

Deep learning (DL) has gained tremendous popularity over the past decade chiefly due to its outstanding performance on computer vision (CV) tasks. Alexnet [1] demonstrated the potential of using gradient descent to train CNNs for image classification which led to a number of deeper CNNs being proposed over the subsequent years. The idea has been that increasing the depth (number of layers in the network) generally increases accuracy since deeper networks have more parameters and more non-linearities and can thus represent a more complex function. In earlier days, increasing the depth beyond a certain point destabilized training because of diminishing gradients but the introduction of skip-connections [2] has alleviated this problem and made it possible to greatly increase the depth of the networks.

Consequently, neural networks are getting deeper and performance has generally improved with depth. AlexNet [1] (23 layers) achieved 15.3%, VGG-19 [3] (43 layers) achieved 7.3%, ResNet-152 [2] (464 layers) achieved 5.5%, and DenseNet-201 [4] (606 layers) achieved 6.34% top-5 error rate on the Imagenet validation set.[1] He, Zhang, Ren, *et al.* [2] and Huang, Liu, Van Der Maaten, *et al.* [4] report even deeper networks with 1,001 and 1,202 layers. Neural networks are also getting wider i.e. having more convolutional filters at each layer or having multiple convolutional layers of various kernel sizes at each level. Googlenet/Inception-v1 [5] (59 convolutions arranged in 22 levels) achieved 6.67% and Inception-v3 [6] (94 convolutions arranged in 42 levels) achieved 4.2% top-5 error rate on the Imagenet validation set. A straightforward consequence of constructing deeper or wider networks is requiring more GPU memory.

GPUs have tried to evolve with the growing need for memory and are getting larger memory sizes. The largest Fermi GPUs had 6 GB RAM, Kepler had 12 GB, Maxwell had 24 GB, Pascal had 16 GB, Volta had 32 GB, Turing had 24 GB, and Ampere had 80 GB. Yet despite this growth in GPU memory size, neural networks have continued to evolve in a memory hungry fashion.

[1]↑Different authors count layers differently. Here, we count all fully connected, convolutional, activation function, normalization, pooling, and dropout layers as distinct.

Neural networks use the vast majority of their memory for one of two purposes. Back-propagation requires storing the intermediate values of the network, i.e., the activations, during the forward sweep for use during the reverse sweep. As networks get deeper, there are more intermediate values. Beyond this, as networks get deeper they tend to have larger model order, i.e., numbers of parameters or weights, not only because they get deeper but also because they get wider.

Neural networks are getting higher model order. AlexNet [1] had 60 million parameters, VGG-19 [1] had 144 million, BERT large [7] had 340 million, GPT-2 [8] had 1.5 billion, and GPT-3 [9] had 175 billion.

Though there is also an opposing counter-trend that argues for reducing model order. While it is an interesting problem to reconfigure a network to give the same performance with fewer parameters as a network with more parameters, in Scorch we want to expand the depth of the network to astronomical scale, where it won't be possible to have so few parameters that can fit on the GPU RAM.

Training deep networks using gradient descent requires computing the gradients of the model parameters with respect to a scalar loss value. This is done using reverse-mode automatic differentiation (AD). Reverse-mode AD saves the output of each intermediate computation (i.e. output of each layer in the network) during the forward sweep into a data structure called the 'tape'. The tape is then consumed in the reverse order during the reverse sweep to compute the gradients of the model parameters with respect to (w.r.t.) the loss. Naturally, increasing the depth or width of the network means increasing the size required to store the tape since more layers mean more intermediate activations or wider layers mean bigger intermediate activations. This imposes a limit on the maximum depth and width of the network because GPU RAMs are limited and can only store the model parameters and the tape upto a certain size. This limitation restrains DL practitioners to experiment with networks of arbitrary width and depth and has caused the performance on various CV tasks to stagnate recently as the maximum achievable network size on current GPUs has been maxed out.

One workaround to this issue is that instead of using a large batch size (say X), use a batch size of 1 in order to minimize the size of the tape and do X number of forward and reverse sweeeps, averaging the gradients of the model parameters computed during each reverse sweep and then update the parameters. This is computationally equivalent to using a batch size of X and makes the tape smaller, allowing a deeper network. However the network size is still limited to whatever size tape can fit on the GPU RAM with a batch size of 1, which is 2400 layers in case of Resnet. An even bigger problem is that the model parameters have to be hosted on the GPU RAM as well and the total size of the parameters keeps going up as the network gets deeper. This reduces the GPU memory available for storing the tape and after a certain point, just the parameters become too big to be hosted on the GPU by themselves.

The current state of the art GPU is NVIDIA's Quadro RTX which offers 48GB of RAM. But the more commonly used GPUs have only 12GB of RAM which is barely sufficient to train Resnet-152 with a batch size of 60. One could use multiple GPUs for training bigger networks by doing 'model parallelism' i.e. putting a chunk of the model on each GPU and transmitting the forward activations and the backward gradients across GPUs. This has shown to be very inefficient because GPUs sit idle when waiting for other GPUs to finish their job and if the network is so big that it requires tens or hundreds of GPUs, this scheme becomes infeasible. Instead people use 'data parallelism' to leverage multiple GPUs by putting a replica of the model on each GPU and synchronize the gradients to perform the same model update thus allowing an overall bigger batch size to be used which eases training and reduces convergence time. This brings us back to the limitation that the tape for the entire network should be able to fit on 1 GPU with atleast a batch size of 1.

Here, we introduce two novel and crucial methods for reducing the GPU memory requirements of extremely deep and extremely wide neural networks. The first is an approach to *divide-and-conquer checkpointing* [10] generalized to GPU computation performed by arbitrary differentiable programs. This reduces the memory requirement for storing the activations. The second is an approach to *tensor streaming* that performs just-in-time migration of data back and forth between the CPU and GPU in parallel to GPU computation. This allows CPU memory to serve as as kind of virtual memory for the GPU RAM. This is important because while the largest currently available GPUs have 80 GB of RAM, current CPUs can contain as much as 8 TB of RAM in a single node.

Our methods have been implemented in a system called Scorch.[2] Scorch is a dialect of Scheme that has support for GPU computation based, in part, on the Torch library [11]. Almost all deep-learning frameworks, like Torch [11], Caffe [12], MXNet [13], Chainer [14], TensorFlow [15], Theano [16], DarkNet [17], and PyTorch [18], implement backpropagation either in a limited domain-specific language or embedded in an existing programming-language implementation through a foreign-function interface. The techniques of divide-and-conquer checkpointing and tensor streaming, particularly when generalized to support not only deep neural networks but also arbitrary differentiable programming, cannot be implemented in such a fashion because they require specialized low-level support from the programming-language implementation and run-time system. Thus Scorch is not based on any existing Scheme implementation but rather on a custom implementation that provides the requisite low-level support.

Scorch is a 'middle of the road' artifact. Most recent publications on differentiable programming within the POPL, PLDI, and ICFP communities [19]–[28] either do not come with any implementation at all, or if they do, only come with one that does not run on GPUs or is incapable of running real-world deep-learning applications at competitive speeds. In contrast, our implementation has sufficient functionality that it can run real-world computer-vision applications like ResNet for image classification (Section 6.1) and DRANet for semantic segmentation (Section 6.2) and real-world natural-language processing applications like the GPT-3 transformer language model (Section 6.3), taking gradients of large long-running computations on the GPU.

Scorch can run ResNet-152 with a network of the same depth and model order as the corresponding PyTorch implementation almost as fast as PyTorch. But it can also run variants of this network with three orders of magnitude greater depth and model order, something that neither PyTorch nor any other current system can do, and do so with essentially the same speed relative to the size of the network as the smaller network. As illustrated in Section 6, Scorch supports running on a single GPU, multiple GPUs per node, and multiple nodes with multiple GPUs, using Infiniband for connection between nodes. On the other hand, Scorch is a research prototype, not a production artifact. It lack the huge

[2]↑We will release code at `https://github.com/qobi/scorch` upon acceptance.

set of builtin features of a system like PyTorch. Our hope is that others will use our code as the basis of a future production artifact.

While there has been considerable recent work on both checkpointing and tensor streaming, as we review in Chapter 7, our work here differs from this work in three key ways. First unlike all recent work, it supports divide-and-conquer checkpointing and tensor streaming for arbitrary differentiable programs performing tensor and GPU computation, not just neural networks. Second, as discussed in Chapter 4, it supports both divide-and-conquer checkpointing and tensor streaming in a synergistic fashion with optimizations that are only possible when these techniques are combined. Third, as discussed in Chapters 4 and 7, and demonstrated in Chapters 6 and 7.5, these techniques are complementary; one alone will not scale to support extremely deep neural networks and extremely long-running differentiable programs, yet their combination does.

2. PRELIMINARIES

2.1 Overview of Automatic Differentiation

Any particular execution of a program f with a particular control flow can be viewed as a composition of machine-state transition functions $f_1, \ldots, f_T$

$$f = f_T \circ \cdots \circ f_1$$

where x_0 is the input machine state and each x_t is the intermediate result produced by f_t. This can be written as a straight-line single-assignment program:

$$\begin{gathered} x_1 = f_1(x_0) \\ \vdots \\ x_T = f_T(x_{T-1}) \end{gathered}$$

The chain rule

$$f(x_0) = f_T(x_{T-1}) \times \cdots \times f_1(x_0)$$

gives a method for computing the derivative of f assuming known derivatives of the primitive machine-state transition functions f_t. Since each machine state $x_t : \mathbb{R}^n$ is a vector, the machine-state transition functions $f_t : \mathbb{R}^n \to \mathbb{R}^n$ are functions from vectors to vectors, the derivatives $f_t : \mathbb{R}^n \to \mathbb{R}^{n \times n}$ are functions from vectors to Jacobian matrices, i.e., matrices of partial derivatives, and $\times$ is matrix-matrix multiplication. This would require space to store the intermediate Jacobian matrices that is quadratic in the machine-state size n. To alleviate this, instead of computing Jacobians, one computes Jacobian-column-vector products:

$$f(x_0) \times \acute{x}_0 = f_T(x_{T-1}) \times \cdots \times f_1(x_0) \times \acute{x}_0$$

Right-associating this as

$$f(x_0) \times \acute{x}_0 = (f_T(x_{T-1}) \times \cdots \times (f_1(x_0) \times \acute{x}_0))$$

allows computing a Jacobian-column-vector product where the intermediate values take space $O(n)$. This can be written as a straight-line single-assignment program:

$$\begin{aligned}
x_1 &= f_1(x_0) \\
\acute{x}_1 &= f_1(x_0) \times \acute{x}_0 \\
&\vdots \\
x_T &= f_T(x_{T-1}) \\
\acute{x}_T &= f_T(x_{T-1}) \times \acute{x}_{T-1}
\end{aligned}$$

In the above, the values $\acute{x}$ are known as the *tangents* of the associated *primal* values x. Taking $\acute{x}_0$ to be basis vectors allows computing the entire Jacobian matrix, one column at a time. In practice, the machine-state transitions functions f_t are sparse, reading only a small portion of the machine state x_{t-1} and changing only a small portion of the machine state x_t, the operations $f_t(x_{t-1}) \times \acute{x}_{t-1}$ are coalesced into stepwise Jacobian-column-vector-product functions associated with the machine-state transition functions, and these are also sparse. The above is the essence of what is known as *Automatic Differentiation in forward-accumulation mode*, or *forward AD* [29]. Note that the variables x_{t-1} and $\acute{x}_{t-1}$ are live only when computing f_t and f_t. Thus, as formulated above, forward AD takes $O(n)$ space, the same as computing the primal.

One can also compute a row-vector-Jacobian product, equivalently a Jacobian-transpose-column-vector product.

$$\grave{x}_T \times f(x_0) = \grave{x}_T \times f_T(x_{T-1}) \times \cdots \times f_1(x_0)$$
$$f(x_0)^\top \times \grave{x}_0^\top = f_T(x_{T-1})^\top \times \cdots \times f_1(x_0)^\top \times \grave{x}_0^\top$$

Left-associating the former or right-associating the latter as

$$\begin{aligned}\grave{x}_T \times f(x_0) &= ((\grave{x}_T \times f_T(x_{T-1})) \times \cdots \times f_1(x_0))\\ f(x_0)^\top \times \grave{x}_0^\top &= (f_T(x_{T-1})^\top \times \cdots \times (f_1(x_0)^\top \times \grave{x}_0^\top))\end{aligned}$$

can also be formulated as a straight-line single-assignment program:

$$\begin{aligned}x_1 &= f_1(x_0)\\ &\vdots\\ x_T &= f_T(x_{T-1})\\ \grave{x}_{T-1}^\top &= f_T(x_{T-1})^\top \times \grave{x}_{T-1}^\top\\ &\vdots\\ \grave{x}_0^\top &= f_1(x_0)^\top \times \grave{x}_1^\top\end{aligned}$$

In the above, the values $\grave{x}$ are known as the *cotangents*. Taking $\grave{x}_T$ to be basis vectors allows computing the entire Jacobian matrix, one row at a time. In practice, the operations $f_t(x_{t-1})^\top \times \grave{x}_t^\top$ are coalesced into sparse stepwise row-vector-Jacobian-product functions associated with the machine-state transition functions. The above is the essence of what is known as *Automatic Differentiation in reverse-accumulation mode*, or *reverse AD* [30] and generalizes backpropagation [31]. Unlike forward AD, where the computation of the values x and $\acute{x}$ can be interleaved, reverse AD separates them into two *sweeps*: a forward sweep that computes the values x and a reverse sweep that computes the values $\grave{x}$. Crucially, the values x are live throughout the forward sweep and into the reverse sweep when they are consumed by the reverse sweep in reverse order. These saved values are sometimes known as the *tape*. The need to save a tape means that reverse AD, as formulated above, takes $O(nT)$ space.

Real programs do not have a machine state of a fixed size n. They compute a function $f : \mathbb{R}^n \to \mathbb{R}^m$ whose Jacobian is an $m \times n$ matrix, where often $m = 1$, i.e., a gradient. In this case, when $m = 1$ and the primal computation takes $O(T)$ time, a fundamental tradeoff arises: forward AD requires a factor of $O(1)$ more space and a factor of $O(n)$ more time to compute a gradient than to compute the primal, while reverse AD requires a factor of $O(T)$ more space and a factor of $O(1)$ more time. We refer to this as the space/time *penalty* of AD. In particular, reverse AD incurs a *linear* space penalty $O(T)$ in the running time of the program, equivalently neural-network depth. To allow the user to chose whichever is appropriate to the circumstances, SCORCH exposes both forward and reverse AD to the user with two simple primitives:

$$\begin{aligned}\texttt{j*}(f, x, \acute{x}) &= \langle f(x), f(x) \times \acute{x} \rangle \\ \texttt{*j}(f, x, \grave{y}) &= \langle f(x), f(x)^\top \times \grave{y}^\top \rangle\end{aligned}$$

2.2 Overview of Checkpointing

Griewank [10] presented a method to trade off recomputation for reduced memory requirements of reverse AD. Suppose one could decompose a function $f = h \circ g$ with the following straight-line single-assignment program with an intermediate value $v_0 = u_S$:

$$\begin{aligned}u_1 &= g_1(u_0) \\ &\vdots \\ u_S &= g_S(u_{S-1}) \\ v_0 &= u_S \\ v_1 &= h_1(v_0) \\ &\vdots \\ v_T &= h_T(v_{T-1})\end{aligned}$$

Reverse AD would do what appears below on the left. Instead one could do what appears below on the right.

$$
\begin{aligned}
u_1 &= g_1(u_0)\\
&\vdots\\
u_S &= g_S(u_{S-1})\\
v_0 &= u_S\\
v_1 &= h_1(v_0)\\
&\vdots\\
v_T &= h_T(v_{T-1})\\
\grave{v}_{T-1}^\top &= h_T(v_{T-1})^\top \times \grave{v}_T^\top\\
&\vdots\\
\grave{v}_0^\top &= h_1(v_0)^\top \times \grave{v}_1^\top\\
\grave{u}_S^\top &= \grave{v}_0^\top\\
\grave{u}_{S-1}^\top &= g_S(u_{S-1})^\top \times \grave{u}_S^\top\\
&\vdots\\
\grave{u}_0^\top &= g_1(u_0)^\top \times \grave{u}_1^\top
\end{aligned}
\qquad
\begin{aligned}
u_1 &= g_1(u_0)\\
&\vdots\\
u_S &= g_S(u_{S-1})\\
v_0 &= u_S\\
v_1 &= h_1(v_0)\\
&\vdots\\
v_T &= h_T(v_{T-1})\\
\grave{v}_{T-1}^\top &= h_T(v_{T-1})^\top \times \grave{v}_T^\top\\
&\vdots\\
\grave{v}_0^\top &= h_1(v_0)^\top \times \grave{v}_1^\top\\
u_1 &= g_1(u_0)\\
&\vdots\\
u_S &= g_S(u_{S-1})\\
\grave{u}_S^\top &= \grave{v}_0^\top\\
\grave{u}_{S-1}^\top &= g_S(u_{S-1})^\top \times \grave{u}_S^\top\\
&\vdots\\
\grave{u}_0^\top &= g_1(u_0)^\top \times \grave{u}_1^\top
\end{aligned}
$$

This trades off recomputing g for not having to simultaneously have the values u_s and v_t be live, thus doubling the time but halving the space.

Imposing an *individual* checkpoint [32] on a program execution will reduce the space needed for the tape by a constant amount and not change its space complexity. But imposing certain kinds of checkpointing *schedules* on a program execution can change its space complexity and lead to *sublinear* space penalty. A *left-branching* checkpoint tree with $O(T)$ branches will reduce the space penalty to $O(1)$ at the expense of a time penalty of $O(T)$ [33]. A *right-branching* checkpoint tree with $O(\sqrt{T})$ branches will reduce the space penalty to $O(\sqrt{T})$ at the expense of a time penalty of $O(1)$ [34]. Note that a right-branching checkpoint tree with $O(1)$ or $O(T)$ branches will not yield sublinear space penalty. We refer to a right-branching checkpoint tree with a space penalty of $O(\sqrt{T})$ as *square-root* checkpointing. A suitably balanced checkpoint tree leads to $O(\lg T)$ space and time penalties [10]. We refer to this as *divide-and-conquer* checkpointing. Different strategies for choosing the split point S and deciding when to terminate the divide-and-conquer recursion lead to different space-time tradeoffs.

Neural networks are often formulated as simple lists of layers and thus it is often easy to partition these lists into the requisite checkpointing schedule. But we wish to be able to impose the requisite checkpointing schedule on an arbitrary differentiable program which may contain conditionals, loops, and function calls that make the correspondence between the program text and the underlying step sequence opaque. Moreover, some systems require that the user annotate program-text intervals that induce checkpoint intervals under the constraint that annotated intervals in the program text not cross constituent boundaries. Under this constraint, it may not be possible to impose the requisite checkpointing schedule. CHECKPOINTVLAD [35] introduced a general-purpose automatic mechanism for imposing the requisite checkpointing schedule on the underlying sequence of program steps induced by an arbitrary differentiable program, without any user annotation, even one whose control flow varies over the course of execution.

SCORCH is based on CHECKPOINTVLAD and exposes divide-and-conquer checkpointing to the user with a simple primitive

$$\texttt{checkpoint-*j}(f, x, \grave{y}) = \langle f(x), f(x)^\top \times \grave{y}^\top \rangle$$

implemented as follows:

base case (when computing $f(x)$ is fast)**:**

$$\langle y, \grave{x} \rangle = \texttt{*j}(f, x, \grave{y}) \quad \text{(step 0)}$$

inductive case:

$$h \circ g = f \quad \text{(step 1)}$$

$$z = g(x) \quad \text{(step 2)}$$

$$\langle y, \grave{z} \rangle = \texttt{checkpoint-*j}(h, z, \grave{y}) \quad \text{(step 3)}$$

$$\langle z, \grave{x} \rangle = \texttt{checkpoint-*j}(g, x, \grave{z}) \quad \text{(step 4)}$$

Note that `checkpoint-*j` is mathematically equivalent to `*j`, differing only in asymptotic space and time complexity. This allows one to apply divide-and-conquer checkpointing to a program simply by replacing calls to `*j` with calls to `checkpoint-*j`.

The key difficulty in implementing the above is step 1, splitting a procedure f into the composition of two procedures g and h. This is accomplished by introducing a general interruption and resumption interface, reminiscent of engines [36]–[38], via three primitives:

$\texttt{primops}(f, x) \mapsto T$	Return the number T of evaluation steps needed to compute $y = f(x)$.
$\texttt{interrupt}(f, x, S) \mapsto z$	Run the first S steps of the computation of $f(x)$ and return a capsule z.
$\texttt{resume}(z) \mapsto y$	If $z = \texttt{interrupt}(f, x, S)$, return $y = f(x)$.

With this, step 1 could be formulated as $S = \lfloor \frac{\texttt{primops}(f,x)}{2} \rfloor$, g could be formulated as $\lambda x.\textsc{interrupt}(f, x, S)$, and h could be formulated as $\textsc{resume}$.

With suitable hardware and operating system support, `primops` could be implemented via a mechanism to time a computation, `interrupt` could be implemented via a timer interrupt, and `resume` could be implemented by a return from interrupt. Scorch implements this interface by converting the program to continuation passing style (CPS) [39]–[42], threading a step count and step limit, along with ordinary and interruption continuations, insertion of step limit checks to call either the ordinary or interruption continuation depending on whether the step limit was reached, and having the capsule passed to the interruption continuation close over the ordinary continuation. This approach allows low-overhead fine-grained checkpointing. Its simplicity relies on the program being purely functional, with no mutation, so computation can be repeated.

For this to work, it must be possible to apply `*j` in the base case, step 0, to functions f that contain `interrupt` or are `resume`. Moreover, it must be possible to apply `*j` in the base case, step 0, to functions f whose domain and/or range are, or can include, capsules z. The divide-and-conquer nature of `checkpoint-*j` further requires that the above interface nest. To handle the recursion in step 3, it must allow interrupting a resumption, i.e., $\textsc{interrupt}(\textsc{resume}, \ldots)$. To handle the recursion in step 4, it must also allow scheduling two interrupts, i.e., $\textsc{interrupt}(\lambda x.\textsc{interrupt}(\ldots), \ldots)$ so that the sooner one happens first and the later one happens upon resumption. Siskind and Pearlmutter [35] describe how all this is done.

3. TENSOR STREAMING

3.1 Motivation

In the previous section we have shown how to reduce the size of the tape for a program of arbitrary length to able to fit in a limited memory budget. We have shown that this comes with an increase in run time as we recompute parts of the tape during the reverse sweep but checkpointing makes it possible to atleast be able to run these large programs and compute gradients.

Checkpointing is sufficient to do gradient computations for programs where the tape is long but there are three scenarios where checkpointing alone is not enough and we need additional techniques to be able to compute gradients in such situations.

3.1.1 Parameterized programs of a high order

Parameterized programs (neural networks being a major example) need memory to store two kinds of objects: the tape, and the parameters of the model also commonly called the weights. These parameterized programs run gradient descent to compute the gradients of these parameters with respect to a loss which is mostly a scalar value. Hence these programs are of the form $\mathbb{R}^m \rightarrow \mathbb{R}^1$ where m is the number of parameters also called as the model order. Reverse mode automatic differentiation the algorithm of choice for gradient computation of these parameters because the loss value is scalar. The gradients of the parameters are computed and then used to update the parameters using various update methodologies for example SGD[43], SGD with momentum[31], AdaGrad[44], RMSProp[45], Adam[46] etc.

A parameterized program of the above kind has the form where each step of the program takes an input and a parameter and computes the output. Increasing the size of such program means increasing the number of these steps leading to an increase in the number of parameters. For the specialized case of neural networks where each step is a neural network layer and each layer takes an input and a parameter(weight), increasing the size means increasing the number of layers which increases the total size of the weights of the network.

Table 3.1. Sizes of weights and tape for a batch size of 30 for ResNets of various depths.

depth	weights (GB)	tape (GB)
152	0.22	8.85
1,500	2.11	86.23
10,000	14.04	576.08
50,000	69.94	2,870.64
100,000	139.66	5,732.41
1,000,000	1,395.87	57,295.71

As we keep increasing the size of these programs, we run into the situation where the parameters alone use up all of the available memory budget, leaving no room for storing the tape even if the tape is very small by virtue of checkpointing. Increasing the size of these programs further leads to the situation where the parameters do not even fit on the available memory budget thus making it impossible to even instantiate such programs.

We demonstrate this for the example of the popular ResNet [2] network. Table 3.1 shows the memory needed by the weights of the network as well as the tape of the network. Note that the for a ResNet with 152 layers, the total of weights and tape will fit on a 12GB GPU, but increasing the number of layers further will cause us to run out of memory. We can fix this by using checkpointing to reduce the size of the tape but notice that for a ResNet with 10,000 layers, even the weights will not fit on a 12GB GPU.

3.1.2 Extremely Large Intermediate Outputs

Checkpointing reduces the length of the tape or the number of nodes in the tape at any given point in the program so we can fit it in a limited memory budget. But when working with neural networks, we often run into situations where the size of each tape node i.e. the size of the output of each intermediate layer is too big to make checkpointing feasible.

This is a common occurence with 3D computer vision models. We demonstrate this with the example of the popular 2D object detection network YOLO[17]. YOLO in its 2D form, takes as input an image and outputs the coordinates and class ids of the detected objects in the image. To convert YOLO to a 3D model which takes as input a video and outputs the

Table 3.2. Dimension and size of the output of the first layer in a YOLO-3D model for various number of input frames, b=batch size, c=number of channels, t=temporal, h=height, w=width

input number of frames	tensor dimension [b, c, t, h, w]	tensor size (GB)
2	[1, 32, 2, 608, 608]	0.088
16	[1, 32, 16, 608, 608]	0.7
32	[1, 32, 32, 608, 608]	1.4
64	[1, 32, 64, 608, 608]	2.8
128	[1, 32, 128, 608, 608]	5.6
300	[1, 32, 300, 608, 608]	13.2

timestamps, spatial coordinates and class ids of detected activity instances in that video, a natural extension would be to replace the 2D convolutions in the stock YOLO model with 3D convolutions. We refer to this model as YOLO-3D.

A consequence of converting YOLO to YOLO-3D is that the output of each layer has an added temporal dimension. This greatly increases the size of each tape node. Table 3.2 shows that as we increase the number of input frames for such a temporal model, even with a batch size of 1, the cost of memory for the output layers starts to become inhibitively large. A typical video is 30 fps, which means that as we start to input about one second of a video, the output of each layer becomes about 1GB in size.

For such scenarios, checkpointing becomes severely constrained because the YOLO model has 153 layers but on a 12GB GPU, a tape of only about 10 tensors can be held. We will need to checkpoint with a very small base case but recall that checkpointing also stores capsules in order to resume an interruption. Checkpointing produces $O(lgT)$ number of capsules, each containing atleast one tensor for resumption, so the tape needs to be even smaller than 10 tensors.

3.1.3 Criss-cross or inter-connected networks

For a program that is of the kind

$$\begin{aligned} y_0 &= f_0(x_0) \\ y_i &= f_i(y_{i-1}), \quad i = (1, 2, \dots n) \end{aligned}$$

where i is strictly an integer sequence of 1 through n, the output of each intermediate step depends only on the output of the previous step. In this case the tape consists of $y_i,\ i = (0, 1, \dots n-1)$ in sequence and checkpointing is able to reduce the length of this tape.

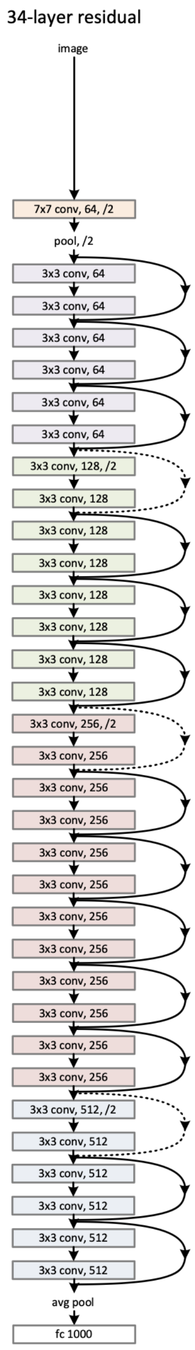
But consider a program of the form

$$\begin{aligned} y_0 &= f_0(x_0) \\ y_i &= f_i(y_{i-1}), \quad i \in [1, n] \end{aligned}$$

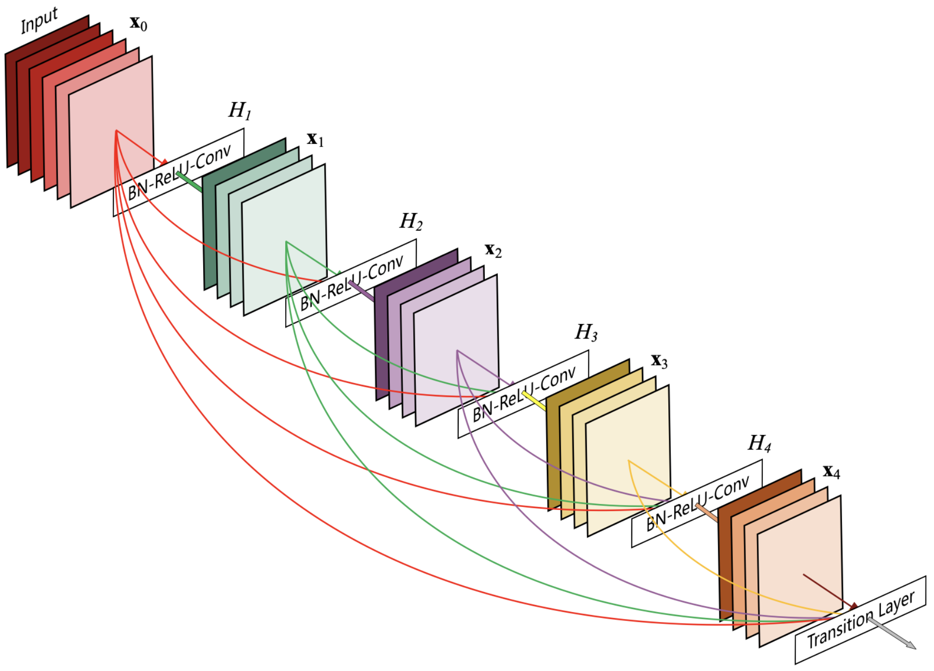
Where i is not a strict sequence, the output of each intermediate step does not necessarily depend on the output of the step right before it, but it could depend on the output of any step that came before it. This is a common occurence in neural network that contain short-cut connections e.g. ResNet[2] and DenseNet[4] as shown in Figure 3.1.

Recall that in checkpointing, all the variables that are required in future computations are stored in capsules. This increases the size of the capsules. The worst case program would do

$$\begin{aligned} y_0 &= f_0(x_0) \\ y_1 &= f_1(y_0) \\ &\vdots \\ y_{n-1} &= f_{n-1}(y_{n-1}) \\ y_n &= f_n(y_0, y_1, \dots y_{n-1}) \end{aligned}$$



(a)



(b)

Figure 3.1. ResNet[2] on the left and DenseNet[4] on the right showing examples of networks with interconnections. While ResNet's interconnections are fairly local, DenseNet's interconnections span the entire network. Reprinted from [2] and [4]

Where the output of the last step uses the outputs of all the previous steps. The outputs of all the previous steps in this case will be stored in capsules and checkpointing, though operating in theory, but won't give us any space saving.

3.1.4 Need for Streaming

The above situations highlight the fact that checkpointing alone doesn't allow running all kinds of programs of arbitrary sizes and number of parameters. Our goal here is to be able to do executions on the GPU which has a very limited memory, but we can leverage the CPU RAM which is normally magnitudes larger in size in order to overcome the above problems. The rest of this section describes a new kind of tensor called the streaming tensor that allows using the CPU RAM as an addendum to the GPU RAM.

3.2 Streaming Methodology

Scorch supports various kinds of tensors: `byte`, `char`, `short`, `int`, `long`, `float`, and `double` that can reside on either the `cpu` or `gpu`. A `cpu` tensor any kind is always manipulated on the `cpu` and a `gpu` tensor of any kind is always manipulared on the `gpu`. We introduce a new class of tensors called `streaming` tensors whose primary residence is on the `cpu` but are always manipulated on the `gpu`. These `streaming` tensors are brought to the `gpu` for manipulation and then evicted when there is a need to free up space. While this migration is reminiscent of virtual memory, we do not use any hardware or operating-system support like memory management units (MMUs) or page faults. The migration is solely handled in software.

We use the following policy for streaming tensors:

1. A `streaming` tensor can reside on both the `cpu` and the `gpu` but it uses the `cpu` as its primary residence. A `streaming` tensor object contains slots for holding a `cpu` copy and a `gpu` copy as well as a flag that mark its `gpu` copy as evictable or non-evictable.

2. A `streaming` tensor can be created from either a `cpu` tensor or a `gpu` tensor upon invocation of `unary_streaming_tensor` on the `cpu` or `gpu` tensor.

3. If it is created from a `cpu` tensor, there is nothing to be done since the tensor's data already resides on the cpu.

4. If it is created from a `gpu` tensor, memory is allocated on the `cpu` and a copy of the data from the `gpu` to the `cpu` is initiated. The `gpu` copy is marked as non-evictable during the copy operation so it is not deleted before the copy operations has finished. When the copy is complete, the `streaming` tensor resides on both the `cpu` and the `gpu` and the `gpu` copy is marked as evictable.

5. When a primitive needs to manipulate a `streaming` tensor and it is only resident on the `cpu`, memory is allocated on the `gpu` and a copy is initiated from the `cpu` to the `gpu`. The `gpu` copy is marked as non-evictable so that it cannot be deleted until the copy operation is finished and until the primitive function that caused this copy has finished using the data. When the copy is complete, the `streaming` tensor resides on both the `cpu` and the `gpu`.

6. When a function needs to manipulate a `streaming` tensor and it is resident on the `gpu` as a results of (4) or (5), the function proceeds and executes. In case of (5), the `gpu` copy is now marked as evictable.

7. Whenever memory needs to allocated on the `gpu` in order to hold the `gpu` copy of a `streaming` tensor, the following are done in order:

    (a) If there is sufficient free space available on the `gpu` then proceed with allocation.

    (b) If there isn't sufficient free space available on the `gpu` then run the garbage collector and repeat (a).

    (c) If there isn't sufficient free space available on the `gpu` then evict all `streaming` tensors from the `gpu` that are marked as evictable and repeat (a).

    (d) If there isn't sufficient free space available on the `gpu` then compact the cache memory and repeat (a).

    (e) If there isn't sufficient free space available on the `gpu` then wait until all CUDA streams have finished execution to see if some `streaming` tensors with `gpu` copies that were non-evictable before are now evictable. Evict these `gpu` copies and repeat (a).

   (f) If there isn't sufficient free space available on the `gpu` then raise an out of memory error.

8. In the default mode of Scorch, the output of a primitive whose inputs were `cpu` tensors is a `cpu` tensor, the output of a primitive whose inputs were `gpu` tensors is a `gpu` tensor and the output of a primitive whose inputs were a mix of `gpu` tensors and `streaming` tensors is also a `gpu` tensor, not a `streaming` tensor.

To make streaming work with AD, we make the change to the AD framework that if we are computing the cotangent of a parameter that is a streaming tensor, we invoke `unary_streaming_tensor` on the cotangent. This converts the cotangent to a streaming tensor and ensures that the gradients of the parameters are also streaming tensors and will not overflow the GPU RAM.

3.2.1 Demand fetch

The above policy suffices running programs with parameters that will not fit on the GPU but will fit on the CPU using algorithm 1.

Algorithm 1 Gradient descent with parameters as streaming tensors

```
1: procedure GRADIENT DESCENT(f, parameters, update_rule, loss_cotangent)
2:     parameters = unary_streaming_tensor(parameters)
3:     loss, parameters_gradient = *j(f, parameters, loss_cotangent)
4:     parameters = update_rule(parameters, parameters_gradient)
5:     return parameters
```

We call this paradigm 'demand-fetch' because the program runs up to the point where when a primitive needs to access a `streaming` tensor which is not resident on the `gpu`, it pauses and waits for the `streaming` tensor to become resident on the `gpu` and then proceeds. Note that this increases the total execution time of the program because it has to wait for the data to be copied from the `gpu` to the `cpu`.

3.2.2 Streaming plan

The NVIDIA GPUs are equipped with DMA controllers and can copy data between page locked CPU memory and GPU memory without interrupting code execution. This allows us to move data between the two memories while the GPU is working on executing other parts of the program.

For static iterative programs with no control-flow, where the access pattern of the tensors will be the same across iterations, there is an opportunity to record the access pattern and overlap the copying of `streaming` tensors from the `cpu` to the `gpu` with GPU execution. To record this access pattern of `streaming` tensors, we introduce a new primitive called `get-streaming-plan`:

`get-streaming-plan`$(f, x) \mapsto p$	Return a streaming plan p, a list of all streaming tensors accessed while computing $f(x)$, in the order they were accessed.

In a fashion similar to `primops`, `get-streaming-plan` takes a procedure f and argument x, where x is a list of all parameters converted to streaming tensors and evaluates $f(x)$, not for the purpose of getting a result, but rather for the purpose of returning the access pattern of the streaming tensors in x. We call this access pattern 'streaming plan'. It is simply a list of `streaming` tensors.

We then introduce another new primitive:

`call-with-streaming-plan`$(p, f, x) \mapsto f(x)$

Evaluate $f(x)$ while running a background thread to opportunistically copy the non-GPU-resident streaming tensors in the streaming plan p, in order, from the CPU to the GPU.

Importantly, the background thread is given a small memory budget called the 'streaming budget' which is a fraction of the total GPU memory and it goes through the streaming plan p implementing the following policy on each tensor.

1. If the streaming tensor is resident on the `gpu`, then do nothing.

2. If the streaming tensor is not resident on the `gpu` then allocate memory within the streaming budget and initiate a copy from the `cpu` to the `gpu`. Mark the `gpu` copy as non-evictable so that it is not deleted during the copy or before the primitive that will be using this tensor from the main thread has finished using it.

3. If sufficient memory is not available to be allocated in (2), then wait and keep inspecting the `streaming` tensors already brought on by (2) to see which ones have been marked as evictable as a result of functions executing in the main thread. Once a tensor is evictable, evict the `gpu` copy. Allocate memory once enough contiguous free space is available.

Note that the background thread works in its own memory budget which is a separate memory pool than that being used by the main thread. Only the background thread is allowed to allocate or free space from within this pool. There is no interaction between the two threads except when functions in the main thread finish execution and mark `gpu` copies of `streaming` tensors to be evictable and the background thread when looking to allocate memory, sees these tensors marked as evictable and evicts them. This methodology ensures that there is no need for synchronization between the main and the background thread and there will be no race conditions.

Note that the above policy still works cohesivly with the demand-fetch framework. Once the background thread starts copying of a `streaming` tensor, it marks it as non-evictable. When the main thread tries to access a tensor and sees it as non-evictable, it can tell that this tensor is either available to be used or in the process of being made available. The main thread then just waits until it is available. This also ensures that the main thread is paused if the background thread is lagging behind so it does not start demand fetching tensors that were going to be made available by the streaming thread in the future. The demand-fetching of a tensor from the main thread only kicks in when the main thread tries to access a tensor that is not being processed by the background thread.

The above scheme of creating a streaming plan and then calling with streaming plan ensures that the parameters are always kept on the CPU but they are eagerly brought on to the GPU in the background and made available to the GPU as soon as it needs them.

Assuming that the copy of tensors is fast and can be completely hidden under GPU execution, we incur no overhead anymore and the program runs as if all parameters were always resident on the GPU.

Algorithm 2 Gradient descent with a streaming plan

```
1: procedure GRADIENT DESCENT2 (f, parameters, update_rule, loss_cotangent)
2:     parameters = unary_streaming_tensor(parameters)
3:     g = lambda(parameters)(Gradient Descent(f, parameters,
                                    update_rule, loss_cotangent))
4:     p = get-streaming-plan(g, parameters)
5:     parameters = call-with-streaming-plan(p, g, parameters)
6:     return parameters
```

3.3 Activation Streaming

We have showed how to solve the issue raised in Section 3.1.1 but we still haven't solved the issues of Section 3.1.2 and Section 3.1.3. The tensor streaming framework described so far is limited to the streaming of parameters and doesn't apply to the intermediate outputs of a program because we have the stipulation that the output of a primitive whose inputs were a mix of `gpu` and `streaming` tensors is a `gpu` tensor.

To resolve this, we lift the above stipulation and provide a new mode in our program that can be enabled and switches to the policy where the output of each `gpu` manipulation is converted to a streaming tensor. This leads to the following additions to the policy of Section 3.2.

9. In the activation streaming mode, we invoke `unary_make_intermediate_streaming_tensor` on the output of primitives whose inputs were `gpu` tensors, `streaming` tensors or a mix of the two. The `unary_make_intermediate_streaming_tensor` creates a streaming tensor but does not initiate a copy of the tensor from the `gpu` to the `cpu`. The tensor is added to a list called 'intermediate tensors'.

10. Whenever we access a tensor that is on the intermediate tensors list, we move it to the bottom. This has the effect that the tensors are sorted top to bottom with respect to their last usage.

Table 3.3. Comparison of execution time, activation transport time, and weight transport time in seconds for some common convolutions used in ResNets. The execution times are calculated by computing the time needed to execute the FLOPs of the layer on a 100 TFLOP/s GPU and the transport times are calculated by computing the time needed to transport a tensor of the given size on a 16 GB/s PCIe-3.0 bus. Actual times may vary due to launch overheads and bus traffic.

input shape	weight shape	execution time	activation transport time	weight transport time
256×56×56	64×256×1×1	1.65×10^{-5}	6.02×10^{-3}	4.09×10^{-6}
64×56×56	64×64×3×3	3.47×10^{-5}	1.50×10^{-3}	9.21×10^{-6}
64×56×56	256×64×1×1	1.67×10^{-5}	1.50×10^{-3}	4.09×10^{-6}
512×28×28	128×512×1×1	1.77×10^{-5}	3.01×10^{-3}	1.63×10^{-5}
128×28×28	128×128×3×3	3.47×10^{-5}	7.50×10^{-4}	3.68×10^{-5}
128×28×28	512×128×1×1	1.78×10^{-5}	7.50×10^{-4}	1.63×10^{-5}

11. When we need to allocate new memory and options 7(a)-(e) are exhausted, we initiate a copy of the tensor at the top of the intermediate tensors list and evict the `gpu` copy. We keep repeating this until there is enough space available or the intermediate tensors list becomes empty at which point we raise an out of memory error.

Activation streaming solves the issues of Section 3.1.1 and Section 3.1.3. We don't make it the default mode of operation of SCORCH because it is generally cost inhibitive to stream large intermediate output tensors. It is generally faster to recompute them, which is what checkpointing does, but slower to transfer them back and forth between the GPU and the CPU. This mode should only be used when all other options have been exhausted. Table 3.3 shows the time required to compute intermediate outputs and the time required to transport them.

3.4 Tensor Streaming vs. CUDA Unified Memory

CUDA supports data migration between the CPU and GPU using *unified memory*, a single address space that is accessible from any CPU and GPU in the system. The user allocates unified memory using a special-purpose function `cudaMallocManaged`. Upon accessing this memory on any CPU or GPU, the CUDA driver automatically migrates the pages to the relevant device and makes them available for access. When a GPU runs low on memory, the CUDA driver migrates old pages back to the CPU to make room for new pages. Without any special-purpose user programming, the migration of memory pages is triggered by page faults, i.e., accessing the data on a device where it is not resident triggers a page fault, the driver halts program execution, migrates the data, and resumes execution. We call this mode 'unified memory no prefetching'.

The CUDA driver is not aware of the size of a tensor so in the no prefetching mode, it only brings on the page that is touched by a GPU kernel. Pages are only 4kB in size so in the no prefetching mode, several page faults need to be triggered and several migrations need to happen in order for the whole tensor to be available. This can be made more efficient by prefetching an entire tensor when a gpu primitive tries to access a tensor by making call to a special-purpose function `cudaMemPrefetchAsync`. We call this mode 'unified memory prefetching' and it is synonymous to our 'demand-fetch' paradigm.

Since we have the functionality to record the access pattern and make a streaming plan for the execution of our program, we can call `cudaMemPrefetchAsync` on the tensors in a background thread much like our `call-with-streaming-plan` primitive. We call this mode 'unified memory prefetching with plan'.

Unified memory behaves similar to tensor streaming, still there are some differences which make tensor streaming more efficient than unified memory. Tensor streaming always keeps a copy of the tensor on the CPU. Thus data copied to the GPU can be evicted after the program is finished using it without having to copy it back to the CPU. Since SCORCH is a functional language, where we do not mutate data, we are always sure that any data copied to the GPU will not be changed and need not be copied back. Unified memory, on the other hand, migrates instead of copies, and has to copy data back to the CPU after processing.

We performed experiments using tensor streaming, unified memory no-prefetching, ,unified memory prefetching and unified memory prefetching with plan to compare their performance (Section 7.5). Tensor streaming performs much better than all three variants of unified memory. Note that both tensor streaming and unified memory have the propensity to fill up GPU RAM by not evicting used data until the memory is full and space is required to copy or migrate new data. We circumvent this issue by limiting the amount of GPU RAM available to these mechanisms. Tensor streaming makes sure that data copied to the GPU does not get evicted until it is used. Unified memory prefetching, on the other hand, can migrate pages back before they are used, to make room for new prefetched pages, which results in on-demand migration of old pages when the program tries to access them. Indeed our experiments showed that unified memory no-prefetching actually performed better than unified memory prefetching.

4. MODIFICATIONS TO CHECKPOINTING

In this chapter, we discuss the crucial modifications to the checkpointing algorithm to make it feasible for use with training deeper neural networks. While Siskind and Pearlmutter [35], and the associated CHECKPOINTVLAD implementation, provided divide-and-conquer checkpointing, it was only a proof-of-concept. It did not support tensors, did not support GPU computation, and was only ever run on a single tiny unrealistic artificial pedagogical benchmark (Siskind and Pearlmutter [35, Figs. 28 and 29]), specifically designed solely to demonstrate that it exhibited the theoretical $O(\lg T)$ space penalty, while TAPENADE (Hascoët and Pascual [47]) did not (Siskind and Pearlmutter [35, Fig. 30]). (TAPENADE exhibits $O(T)$ space penalty on this example.)

Because CHECKPOINTVLAD did not support tensors and GPU computation, it could not be applied to practical deep-learning applications. SCORCH significantly builds upon CHECKPOINTVLAD by adding support for tensor computation that runs both on the CPU and on the GPU. It also adds support for multiple GPUs and multiple nodes. This allows it to be useful and practical for realistic, large deep-learning applications such as ResNet (Section 6.1), DRANet (Section 6.2), and GPT (Section 6.3). The contribution of SCORCH over CHECKPOINTVLAD as an artifact is enormous. The source-code base grew from 8,366 lines to 97,793 lines. The checkpointing algorithm also evolved in substantive ways. We now discuss one such evolutionary enhancement that is crucial to running the examples shown later.

Machine learning often iterates a simple gradient-descent step to update the model parameters θ. The gradient $\nabla_\theta f(\theta, x)$, denoted as $\grave{\theta}$, can be computed with `*j` (or `checkpoint-*j`).

$$\begin{aligned}\langle l, \grave{\theta}\rangle &= \texttt{*j}(f, \theta, 1)\\ \theta &:= \delta(\theta, \grave{\theta}, \eta, m)\end{aligned}$$

Here, $l = f(\theta)$ is the loss, δ is the update function which could be any of SGD, SGD with momentum, AdaGrad, AdaBoost, Adam etc. η are the hyper-parameters required by the update function and could be the learning-rate, momentum etc. m is the state of the parameters. Different optimization methods use different state variables for example, SGD with momentum uses the velocity of the parameters, Adam uses the first and second moments, etc.

When the model order is high, the model parameters θ do not fit in GPU RAM. The gradients of the parameters also do not fit in GPU RAM since they are the same size as the parameters. As described in Section 3.2, the parameters are kept on the CPU and only brought to the GPU for manipulation, and the computed cotangents are also converted to streaming tensors immediately and shipped to the CPU. This means that when we are about to perform the update step, the parameters θ and their gradients $\grave{\theta}$ reside on the CPU.

We thus have two choices: we can either perform the update step on the CPU or copy both the model parameters θ and their gradients $\grave{\theta}$ back to the GPU, perform the update step on the GPU, and copy the result back to the CPU. Both of these methods are inefficient: the former because the CPU takes longer to perform the update computation than the GPU and the latter because it incurs twice the communication cost.

To enable a more efficient solution, we introduce new primitives that fuses the update step into the gradient computation

$$\langle l, \theta \rangle = \texttt{*j-update}(f, \theta, \grave{l}, \delta(\eta), m)$$
$$\langle l, \theta \rangle = \texttt{checkpoint-*j-update}(f, \theta, \grave{l}, \delta(\eta), m)$$

where $\theta = \delta(\theta, \grave{\theta}, \eta, m)$. Crucially, such fusion allows the gradient computation to be specialized so that the last step of the gradient computation is interleaved with the update step. Thus the last step of the gradient computation does not initiate copying the model-parameter gradients $\grave{\theta}$ back to the CPU allowing their eviction, but rather copies the model parameters θ to the GPU, if needed, during the last step of the gradient computation. Since the model-parameter gradients $\grave{\theta}$ are not exposed as output of this primitive, they become dead upon completion and can be evicted without copying to the CPU. The model-parameter gradients $\grave{\theta}$ are never allocated explicitly as a tensor, streaming or otherwise, and only exist ephemerally as the internal intermediate values inside the `*j-update` primitive. This allows performing the update step on the GPU and alleviates the need to explicitly represent and transport the model-parameter gradients.

One further provision we need in order to make AD work with tensor streaming is in-place update of parameters. For the specialized case of machine learning optimization algorithms, the parameters are only updated after the gradient computation has been performed. The new parameters are used for the next iteration whereas the old parameters are discarded. This gives us the opportunity to reuse the memory of the old parameters for the new parameters. This is actually crucial to make the streaming paradigm work with AD because recall that a streaming plan is a list of tensors in their order of execution. If we discard the old parameters, we are discarding the tensors on the streaming plan. We would then need to create a new streaming plan in order to record the access of the new tensors. Instead, we make two more variants of the AD operators

$$\langle l, \theta \rangle = \texttt{*j-update!}(f, \theta, \grave{l}, \delta(\eta), m)$$
$$\langle l, \theta \rangle = \texttt{checkpoint-*j-update!}(f, \theta, \grave{l}, \delta(\eta), m)$$

The ! indicates that the update step does a side-effect i.e. it performs the update in place so that the streaming plan remains valid across iterations and we do not have to recreate the streaming plan before each iteration.

Neural networks are a restricted form of program formed by composing specific kinds of layers in specific kinds of computation graphs. Many deep-learning frameworks are domain-specific languages (DSLs) for this restricted form of program. Backpropagation in neural networks is a restricted form of reverse AD for this restricted form of program. Many deep-learning frameworks only support backpropagation for this restricted form of program. In contrast, Scorch supports differentiable programming; it supports reverse AD (as well as forward AD) of arbitrary programs. (Extremely) deep neural networks are a restricted form of (extremely) long-running programs. Some deep-learning frameworks only support checkpointing for neural networks. Further, some support only limited forms of checkpointing that do not lead to sublinear space penalty and do not support divide-and-conquer checkpointing. In contrast, Scorch supports divide-and-conquer checkpointing to allow taking gradients of arbitrary extremely long-running programs, not just extremely-deep neural networks. Furthermore, prior frameworks, like Tapenade, that supported differentiable programming, including the ability to apply both forward and reverse AD, as well as limited forms of divide-and-conquer checkpointing, to arbitrary programs, lacked support for tensors and GPU computation, rendering them ill-suited to deep learning. In contrast, Scorch supports tensors and GPU computation and is well suited to deep learning, being able to employ divide-and-conquer checkpointing for taking gradients through extremely deep neural networks as a special case. We illustrate this with our ResNet (Section 6.1), DRANet (Section 6.2), and GPT (Section 6.3) examples.

Ultimately, the efficiency of a system that performs divide-and-conquer checkpointing depends on the efficiency of the underlying interruption and resumption mechanism that is used to break the execution of an arbitrary program up into intervals to impose the requisite balanced nested checkpointing schedule onto the execution of that program. Our experiments in Section 6 evaluate the run times while changing the base-case duration, i.e., changing the depth of the divide-and-conquer checkpointing recursion, to make the requisite size of the tape (i.e., the length of the red and blue lines and the number of violet lines in Figure **??**) longer or shorter. This measures the overhead of the interruption and resumption mechanism.

5. THE SCORCH LANGUAGE

The Scorch language is nominally a subset of R^{4}RS [48] with additions for differentiable programming. It is based on the VLAD language [35], [49]. The syntax of the user interface is very much like that of Scheme, a widely known programming language in order to facilitate users to rapidly pickup and use Scorch. The back end of Scorch is written in C.

5.1 Types

Scorch uses a single polymorphic type called `thing` which can take on any of the following types: `true`, `false`, `null`, `char`, `string`, `bundle`, `tape`, `pair`, `closure`, `capsule`, `data-loader` or ⟨*residence*⟩`-`⟨*type*⟩`-tensor` where ⟨*type*⟩ denotes one of the types `byte`, `char`, `short`, `int`, `long`, `float`, or `double` and ⟨*residence*⟩ denotes one of the residences `cpu`, `gpu`, or `streaming`. Type checking is performed dynamically during execution to call the specialized type variant of the primitive functions.

5.2 Tensor Support

To create `cpu-`⟨*type*⟩`-tensors`, our C backend interfaces with the Torch[11] TH library to access the following two structures.

```
typedef struct THStorage
{
    real *data;
    ptrdiff_t size;
    int refcount;
    char flag;
    THAllocator *allocator;
    void *allocatorContext;
    struct THStorage *view;
} THStorage;

typedef struct THTensor
{
    long *size;
    long *stride;
    int nDimension;
    THStorage *storage;
    ptrdiff_t storageOffset;
    int refcount;
    char flag;
} THTensor;
```

The THStorage struct holds the memory and a reference count of the storage. The THTensor struct points to a THStorage and contains the nDimension, size and stride fields that allow viewing the memory as a multi-dimensional object. These structs are expanded via macros to generate code for the types `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, and `Double`.

To create `gpu-`⟨*type*⟩`-tensors`, our C backend interfaces with the TORCHTHC library to access the complementary versions of the above two structs for CUDAmemory and are expanded via macros to generate code for the types `CudaByte`, `CudaChar`, `CudaShort`, `CudaInt`, `CudaLong`, `Cuda`, and `CudaDouble`.

A SCORCH object 'thing' just wraps around such Tensor objects.

Note that there is nothing magical about the TH and THC interfaces and we could have made our own memory objects but we utilized TORCHś already mature libraries to save time.

5.3 Basic Primitives

SCORCH has the standard unary basis procedures `sqrt`, `exp`, `log`, `sin`, `cos`, `zero?`, `positive?`, and `negative?` that work on scalars as well as pointwise on tensors of all types and residences.

SCORCH has the standard binary basis procedures `+`, `-`, `*`, `/`, `max`, `min`, `atan`, `=`, `<`, `>`, `<=`, and `>=` that work on scalars, a tensor and a scalar, or a scalar and a tensor, or two tensors of the same type and dimensions, in a pointwise fashion.

For the case of tensors, all of the above primitives are wrappers around standard functions from the TORCH library. Again, there is nothing magical about the TORCH library and this was done for the sake of not reinventing the wheel.

Table 5.1. Main SCORCH tensor primitives. ⟨*type*⟩ denotes one of the types `byte`, `char`, `short`, `int`, `long`, `float`, or `double`. ⟨*floating type*⟩ denotes one of the types `float` or `double`. ⟨*residence*⟩ denotes one of the residences `cpu`, `gpu`, or `streaming`.

`list->`⟨*residence*⟩`-`⟨*type*⟩`-tensor`	`fill-`⟨*residence*⟩`-`⟨*type*⟩	⟨*residence*⟩`-`⟨*type*⟩`-tensor?`
⟨*residence*⟩`-`⟨*type*⟩	`randn-`⟨*residence*⟩`-`⟨*floating type*⟩	`normal-`⟨*residence*⟩`-`⟨*floating type*⟩
`size`	`tensor->list`	`view`
`transpose`	`narrow-tensor`	`expand-tensor`
`concat-tensors`	`dot`	`sumall`
`addmv`	`addmm`	`baddbmm`
`addr`	`resize`	`pad`
`crop`	`decimate`	`interpolate`
`interpolate-to-size`	`upsample-nearest`	`downsample-nearest`
`permute`	`ReLU`	`LeakyReLU`
`GeLU`	`sigmoid`	`convolve`
`transpose-convolve`	`batch-normalization-training`	`batch-normalization-test`
`initialize-batch-normalization`	`layer-normalization`	`convolve-add-tied`
`embedding`	`max-pool`	`average-pool`
`dropout`	`dropout-planewise`	`max-value`
`index-of-max`	`cross-entropy-loss`	`softmax`
`fused-scale-mask-softmax`		

5.4 Deep Learning Primitives

Table 5.1 presents the main basis procedures that operate specifically on tensors. There are a few more that do not concern us here. Most are simple wrappers around TORCH[11] and cuDNN [50]. Some are our own implementation in CUDA [51]. And some are implemented in SCORCH as user code in a standard library. While this set of tensor procedures is tiny compared with that in PYTORCH, it is sufficiently rich to allow implementing all of the examples in Section 6.

Most SCORCH basis procedures can take one or more tensors as input and yield a tensor as output. Tensors have a variety of types (`byte`, `char`, `short`, `int`, `long`, `float`, or `double`) and a variety of residences (`cpu`, `gpu`, or `streaming`). SCORCH basis procedures generally require all input tensors to have the same type and residence, and produce output tensors of the same type and residence as the input tensors. They generally also allow a mix of `gpu` and `streaming` tensors as input, producing a `gpu` tensor as output. Primals and associated tangent or cotangent values are not specifically constrained to have the same type and residence. Any constraint between the mutual types and residences of primals and their associated tangents and cotangents results from the fact that the Jacobian-column-vector-product and row-vector-Jacobian-product functions are implemented either as primitives or as compositions of other basis functions and the constraints follow from the properties of this implementation.

5.5 Garbage Collection

Scorch is a functional programming language and we use Boehm garbage collector (GC) [52] to reclaim dead objects and their associated memory. The use of GC greatly simplifies the programming of the backend. The GC periodically examines the state of the program and frees up memory allocated to the pointers that are no longer live.

GC is periodically automatically invoked according to some parameters or criterions internal to the GC. Most likely it is triggered based on how much of the heap has been used up. Since we mostly work with cudamemory and the GC has no notion of how much memory is left on the GPU, we have to call GC manually in the scenarios where we need to allocate GPU memory but there is no free memory available.

When we create tensor objects, we put the calls to free the tensors in the finalizers of these objects so when the objects are collected by GC, the finalizers are invoked which make the appropriate calls to the underlying Torch library to free the associated memory.

5.6 Asynchronicity on the GPU

Scorch very carefully only uses GPU operations that can be launched asynchronously i.e. the control is returned to the CPU once a kernel is launched. This allows the CPU to continue execution and stay ahead of the GPU. The CPU tries to quickly fill up the GPU hardware FIFO queue in order to keep it fully loaded and reach maximum performance. Since most GPU operations are done on tensors which can take a long time, this has the benefit that all CPU execution is practically hidden under GPU computations. The only inevitable synchronization is when we have to copy data from the GPU to get the output of a program e.g. the loss value. But such calls are few and far between, and an appropriately written Scorch program runs at close to 100% GPU utilization.

5.7 Multi-GPU and Multi-Node capability

It is common in deep learning applications to leverage multiple GPUs on the same node or on multiple nodes for the same program to speed up training. There are two main paradigms in which multiple GPUs are leveraged:

- Data Parallelism: Each GPU contains the exact same copy of the network and its parameters. The input data is split along the batch dimension so that each GPU has a chunk of the input batch. Each GPU runs the forward pass, computes the loss and runs the backward pass to compute the gradients of the parameters with respect to its loss. Each GPU broadcasts its gradients to all other GPUs so that each GPU gets an average of the gradients across all GPUs. The averaged gradients are then used to update the parameters on each network.

- Model Parallelism: The network is split across multiple GPUs. The first GPU executes its part of the network and passes the output to the next GPU. The process continues until the last GPU computes the loss and then the backward pass is initiated in the reverse direction. This paradigm is mostly inefficient since GPUs sit idle except the one that is executing its part of the network.

Scorch provides the ability to do data parallelism. We do so by launching multiple processes of the same program through OpenMPI[53]. Each process runs unhindered by other processes, the only point of communication between them is when the gradients need to be averaged. Gradient averaging is done using the NVIDIA NCCLlibrary[54] which gathers, averages and broadcasts the averaged gradients to all processes. This extends to multi-node cases, multiple processes can be launched on multiple nodes and Scorch seamlessly handles gradient averaging across all processes.

Note that each process runs independently with its own data and address space. But we often run into situations where we want to run a very deep network and use tensor streaming to host the parameters on the CPU RAM. But CPU RAM is also limited and doesn't allow as many copies of the parameters to be created on the CPU RAM as the number of processes. For such cases, all the processes on a given node use shared memory. The master process on each node allocates a chunk of shared memory using `mmap` and all the slave processes on that node map this shared memory into their address space. The master process then creates the parameters inside this shared memory and they automatically become available to all the slave processes. While doing gradient computation, the slave processes broadcast their gradients during the gradient averaging step, but then discard them, only the master process retains a copy of the gradients and performs the update of the parameters.

5.8 CUDA Streams

CUDAprovides the notion of streams which are work queues. Work submitted to a specific stream is always executed in order. Work submitted to two different streams can be executed in parallel, depending on the availability of GPU resources. Care has to be taken that work that is dependent on previous work is always added to the same stream. SCORCH uses four streams to fully utilize parallelism

- execution stream
- asynchronous copy stream
- update stream
- nccl stream

The execution stream is responsible for executing the main program on the GPU. It does all the computation involved in the program.

The asynchronous copy stream is used by the streamer or the data loader thread to move data between the CPU and the GPU. Where there is a dependency that the execution stream will use something from the asynchronous copy stream e.g. the data preloaded by the data loader thread, or where the asynchronous copy stream will use something from the execution stream e.g. the computed cotangents that need to be streamed to the CPU, synchronization between the two streams is carefully inserted to run such operations in the correct order.

The update stream is used to carry out the computations of the update step when using `*j!`, `checkpoint-*j!`, `*j-update!` or `checkpoint-*j-update!` because the updates can be run in parallel while the execution stream is carrying out the remaining reverse sweep.

The nccl stream is used in the multi-GPU case to carry out the reduce operation between the cotangent tensors resident on various GPUs. Again, careful synchronization is inserted between the execution, update and nccl streams to make sure that data is only used by another stream when the previous stream has finished its operation on it.

5.9 Tensor Cache Allocator

The standard way to allocate GPU memory is by making calls to the `cudaMalloc` function and to free memory is by making calls to the `cudaFree` function. However both of these functions are synchronous i.e. they synchronize the CPU and the GPU by waiting for all operations queued to the GPU to finish and then perform the malloc or free operation. This synchronization is expensive since it blocks the host (CPU) thread and in a program where there are thousands of malloc and free calls, the run time easily gets dominated by just these two calls.

PyTorch and TensorFlow deal with this issue by allocating all memory required by the program at the beginning and then reusing it throughout the lifetime of the program. This is not possible in a checkpointed-AD paradigm since tensors of different sizes are live during different checkpointing intervals and dynamic memory allocation and freeing is unavoidable.

To avoid synchronization of the CPU and GPU, we allocate all available GPU memory at the beginning of the program and run our own cache allocator on it. We provide two calls `cacheCudaMalloc` and `cacheCudaFree` that allocate memory or return memory to this pool without causing synchronization.

When doing tensor streaming, we wish to overlap the data migration with GPU execution. The `cudaMemcpy` function which is used to copy memory between the CPU and the GPU is also a synchornized function and cost prohibitive. CUDAprovides a variant called `cudaMemcpyAsync` which asynchronously copies memory between the CPU and the GPU with the condition that the memory allocated on the CPU must be page locked and registered with CUDA. Allocating page locked memory and registering with CUDAis again synchronous and cost prohibitive so we allocate a chunk of CPU memory at the beginning of the program and register it with CUDA and run the same cache allocator on it as well. We provide two calls `cacheHostMalloc` and `cacheHostFree` to allocate memory or return memory to this pool.

Since GPU memory is overall limited and we allocate big chunks as tensors, the memory can quickly get fragmented. When the fragmentation is bad such that there is enough total free space available to service a memory allocation request but the memory is not contiguous, we provide a function called `compactCache` for each of the GPU and CPU cache. The `compactCache` function starts at the top of the cache and moves allocated memory upwards to fill up the free spaces in between. This is done in an asynchronous fashion but is still an expensive operation. However, it rarely gets called in most practical programs.

Further, we have several threads (e.g. main thread, data loader thread, streamer thread), each owning a stream (GPU work queue) and working independently of each other. To avoid synchronization between these threads, we provide each of them with their own cache. Because we only make asynchronous GPU calls in almost all of our primitives, the CPU mostly runs ahead of the GPU and keeps queuing operations on the GPU. This results in the scenario where the CPU frees a tensor and allocates that memory to another tensor even before the function that used that tensor has been executed on the GPU, but since each thread has its independent stream that only works in its own memory pool, that memory is not overwritten by another stream and the relevant stream of that thread executes operations in sequence.

The only exception to this is the data loader which allocates memory in its pool to preload the data but that memory is used by the main thread, and the nccl stream which synchronizes the memory regions across GPUs. Careful stream synchronization is placed in these scenarios to make sure that memory is never overwritten before it has been completely utilized.

5.10 Dynamic execution

Just like PyTorch, Scorch dynamically executes a program i.e. it executes functions as it encounters them unlike TensorFlow which constructs a graph of the computation prior to running it and then executes it. This dynamic execution allows us to put control flow in the program where execution can follow conditionals on runtime, allowing us to write programs such as ray tracers with termination conditions determined on run time.

5.11 Further Synergies

The design and implementation of Scorch is synergistic; all of its features interoperate and play well together.

Divide-and-conquer checkpointing is well suited to a pure functional language. The machine state that needs to be saved upon interruption is reflected in immutable continuations, allowing computation to be repeated and resumed multiple times (Figure **??**f) from the same saved state.

Divide-and-conquer checkpointing and tensor streaming serve complementary purposes. Divide-and-conquer checkpointing reduces the memory requirement for the tape, which corresponds to neural-network activations. Tensor streaming reduces the GPU RAM requirement for model parameters, which correspond to the neural-network weights. One could imagine an alternate design that used static methods for constructing streaming plans based on a fixed program structure or neural-network architecture. Indeed, the access patterns of forward and reverse AD are sufficiently simple that this could be done easily; forward AD accesses the model parameters in the same order for the primal and tangent computation, while reverse AD accesses the model parameters in the reverse order from the forward sweep during the reverse sweep. However the access pattern of divide-and-conquer checkpointing is far more complex and difficult to analyze statically as it depends on the base-case duration and the precise number of virtual-machine instructions that can be executed between interrupts. Using `get-streaming-plan` to dynamically construct a streaming plan through profiling makes this easier. The fact that our design allows tensors to be copied on demand, without a streaming plan, or with an inaccurate one, has several advantages. First, it avoids a chicken-and-egg problem and allows the plan to be constructed in the first place by running code when a plan is not yet available. Second, it allows our system to operate correctly, albeit less efficiently, if one constructs a representative streaming plan for a specific control flow that induces a particular access pattern, but uses it when the dynamic control flow varies and induces different access patterns.

6. EXAMPLES

We evaluate the performance of our implementation on three real-world examples. The Scorch source code for these examples is included in the supplementary material. For all of the examples below, we have written full-fledged implementations that support training, i.e., computation of gradients of the loss function and associated parameter updates. We further implemented data loaders that support training on large real-world datasets. Fully training all of these models to state-of-the-art levels of performance would take an immense amount of time and is not our purpose here. Our purpose here is simply to demonstrate the feasibility of training extremely large variants of these widely-used models, purely from the perspective of fitting these models into GPU RAM. Thus we do not train them to state-of-the-art levels of performance; we only train them for a few iterations to measure run times.

6.1 Image Classification

Our first example is ResNet [2], one of the most prominent and highest performing deep-learning image-classification systems that is widely used in the computer-vision community. What is relevant to our purposes is that the ResNet neural-network architecture is formulated as a cascade of various kinds of blocks whose input and output are tensors of the same size. This allows blocks of the same kind to be repeated by varying amounts to create shallower or deeper networks. The original ResNet paper evaluated variants with 18, 20, 32, 34, 44, 50, 56, 101, 110, 152, and 1,202 layers. Here, we reimplement ResNet in Scorch (Figure 6.1) and evaluate variants with 152, 302, 602, 1,001, 1,502, 3,003, 5,000, 10,001, 20,000, 50,000, 100,001, 150,002, and 250,001 layers, simply by changing the hyperparameters `nblocks1`, `nblocks2`, `nblocks3`, and `nblocks4`, to demonstrate that Scorch allows training extremely deep neural networks, ones that would be impossible to train on any other existing system. The ability to train such extremely deep networks relies crucially on divide-and-conquer checkpointing and tensor streaming.

For our experiments, we train ResNet on the ILSVRC 2012 training set [55] for 20 iterations. Figure 6.2 compares run times for doing this with the Scorch and PyTorch [56]

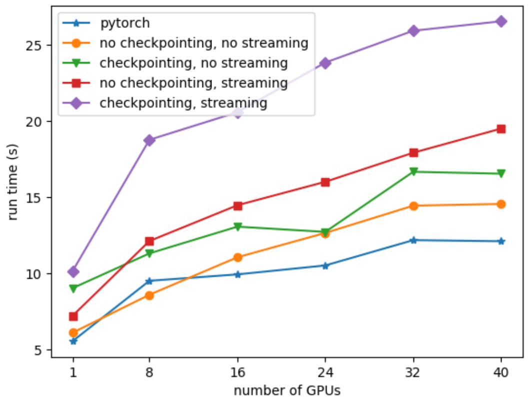
```
(define ((first-block) (list weights-conv weights-bn))
 (sequential-layer
  ((convolution-layer-no-bias '(3 3) (list '(2 2) '(1 1)))
                                              weights-conv)
  ((batch-normalization-training-layer) weights-bn)
  ((ReLU-layer))
  ((max-pool-layer '(3 3) '(1 1) '(2 2)))))

(define (build-blocks i n)
 (if (= i n)
     '()
     (cons (if (zero? i)
               resnet-block-conv-shortcut
               resnet-block-identity-shortcut)
           (build-blocks (+ i 1) n))))

(define (((resnet-block-identity-shortcut)
          (list weights-conv1 weights-bn1
                weights-conv2 weights-bn2
                weights-conv3 weights-bn3))
         x)
 (((ReLU-layer))
  (+ x
     ((sequential-layer
       ((convolution-layer-no-bias '(0 0) (list '(1 1) '(1 1)))
                                               weights-conv1)
       ((batch-normalization-training-layer) weights-bn1)
       ((ReLU-layer))
       ((convolution-layer-no-bias '(1 1) (list '(1 1) '(1 1)))
                                               weights-conv2)
       ((batch-normalization-training-layer) weights-bn2)
       ((ReLU-layer))
       ((convolution-layer-no-bias '(0 0) (list '(1 1) '(1 1)))
                                               weights-conv3)
       ((batch-normalization-training-layer) weights-bn3)) x))))
```

Figure 6.1. The essence of our SCORCH implementation of ResNet

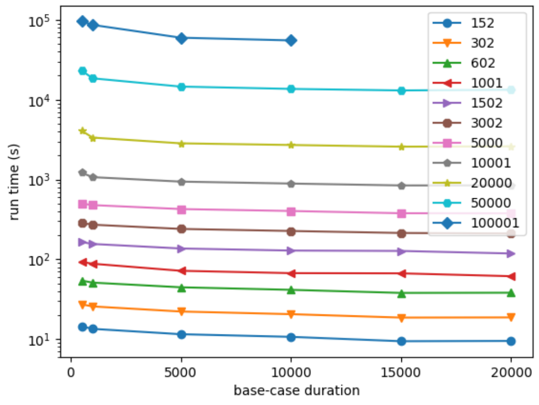
```
(define (((resnet-block-conv-shortcut)
          (list weights-conv1 weights-bn1
                weights-conv2 weights-bn2
                weights-conv3 weights-bn3
                weights-conv-sh weights-bn-sh))
         x)
 (((ReLU-layer))
  (+ ((sequential-layer
       ((convolution-layer-no-bias '(0 0) (list '(1 1) '(1 1)))
                                           weights-conv1)
       ((batch-normalization-training-layer) weights-bn1)
       ((ReLU-layer))
       ((convolution-layer-no-bias '(1 1) (list '(1 1) '(1 1)))
                                           weights-conv2)
       ((batch-normalization-training-layer) weights-bn2)
       ((ReLU-layer))
       ((convolution-layer-no-bias '(0 0) (list '(1 1) '(1 1)))
                                           weights-conv3)
       ((batch-normalization-training-layer) weights-bn3)) x)
     ((sequential-layer
       ((convolution-layer-no-bias '(0 0) (list '(1 1) '(1 1)))
                                         weights-conv-sh)
       ((batch-normalization-training-layer) weights-bn-sh)) x))))

(define ((last-block) (list weights-fc1 biases-fc1))
 (sequential-layer ((average-pool-layer '(7 7) '(0 0) '(1 1)))
                   ((flatten-layer))
                   ((fc-layer) (list weights-fc1 biases-fc1))))

(define (resnet nblocks1 nblocks2 nblocks3 nblocks4)
 (list first-block
       (build-blocks nblocks1)
       (build-blocks2 nblocks2)
       (build-blocks2 nblocks3)
       (build-blocks2 nblocks4)
       last-block))
```

Figure 6.1. Cont. The essence of our Scorch implementation of ResNet.

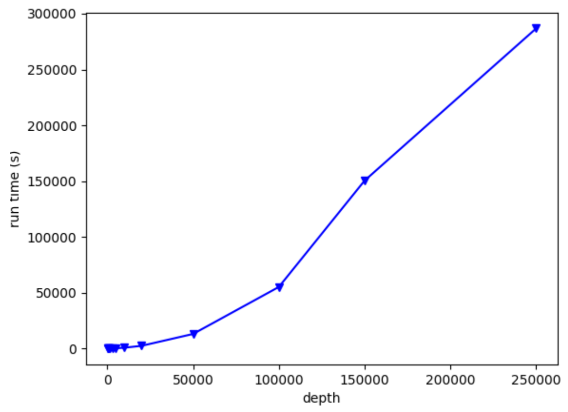
implementations of ResNet under various conditions. Each iteration involves using reverse AD to compute the gradient of a loss function on a batch of 30 images and updating the weights. When n GPUs are used, the batch size is $30n$. Figure 6.2(a) compares the run time of the Scorch implementation of ResNet-152, with and without divide-and-conquer checkpointing, with and without tensor streaming, with the PyTorch implementation, for various numbers of GPUs. The Scorch run time, without divide-and-conquer checkpointing and tensor streaming, is generally the same as the PyTorch run time. Divide-and-conquer checkpointing and tensor streaming generally reduce the speed by no more than a factor of two. Figure 6.2(b) compares the run time of the Scorch implementation, on a single GPU, with divide-and-conquer checkpointing and tensor streaming, for various depths and base-case durations. The run time is generally the same for a given depth, as the base-case duration varies, demonstrating that our implementation incurs little overhead in the interruption and resumption mechanism. Figure 6.2(c) compares the run time of the Scorch implementation, on a single GPU, with divide-and-conquer checkpointing and tensor streaming, for various depths, all with a base-case duration of 10,000. This shows that our implementation can scale to a depth of 250,001. The run time scales slightly superlinerarly due to the $O(\lg T)$ time penalty of divide-and-conquer checkpointing. The largest previous version of ResNet in the original paper has a depth of 1,202; we increase that $200\times$. Figure 6.2(d) compares the run times with the implementation of tensor streaming from Section 3 with an alternate implementation based on NVidia unified memory, both with and without prefetching, where prefetching is used to implement the streaming plan, for the Scorch implementation on a single GPU, with divide-and-conquer checkpointing, and with a base-case duration of 20,000. The implementation of tensor streaming from Section 3 vastly outperforms the alternate one based on NVidia unified memory, irrespective of whether prefetching is used.



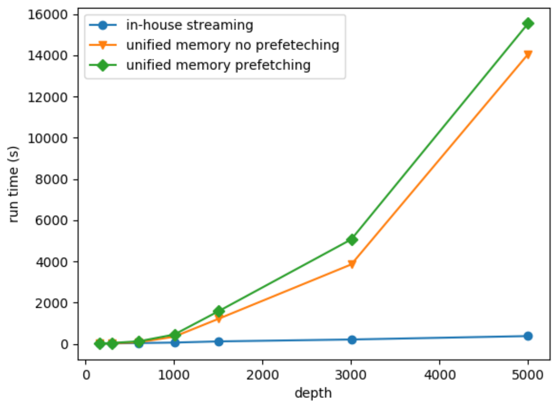
(a)



(b)



(c)



(d)

Figure 6.2. Run times (in seconds) for 20 iterations of ResNets of various depths, with and without divide-and-conquer checkpointing with various base-case-durations, with and without various tensor-streaming methods, and with a batch size of 30 per GPU. There are 8 GPUs per node, so 16 GPUs is over two nodes, 24 GPUs is over three nodes, etc. (a) and (d) use a base-case duration of 20,000. (c) uses a base-case duration of 10,000. A depth of 152, tensor streaming, and a single GPU are used when not specified. Tabular versions in Table 6.1, Table 6.2, Table 6.3, and Table 6.4.

Table 6.1. Tabular version of Figure 6.2(a).

divide-and-conquer checkpointing?	tensor streaming?	number of GPUs 1	8	16	24	32	40
		6.10	8.60	11.05	12.65	14.44	14.56
✓		9.02	11.31	13.06	12.72	16.67	16.54
	✓	7.21	12.12	14.47	16.01	17.92	19.50
✓	✓	10.14	18.77	20.57	23.84	25.92	26.54
PyTorch		5.54	9.51	9.93	10.52	12.18	12.11

Table 6.2. Tabular version of Figure 6.2(b).

	base-case duration					
depth	20,000	15,000	10,000	5,000	1,000	500
152	9.50	9.42	10.69	11.54	13.47	14.45
302	18.73	18.64	20.57	22.21	25.71	27.32
602	38.18	37.97	41.51	44.66	51.06	54.17
1,001	61.49	66.77	67.15	71.90	88.04	92.73
1,502	118.37	127.45	128.82	136.79	155.70	164.55
3,002	210.89	213.95	226.65	240.57	271.47	285.39
5,000	377.18	378.03	403.50	426.58	477.87	499.98
10,001	843.62	842.82	889.73	941.23	1,069.89	1,215.57
20,000	2,602.14	2,578.20	2,706.30	2,839.45	3,350.23	4,077.27
50,000	13,316.80	13,056.80	13,641.62	14,590.75	18,507.70	23,173.54
100,001			55,252.44	59,785.77	86,397.07	95,946.12

Table 6.3. Tabular version of Figure 6.2(c).

depth	152	302	602	1,001	1,502	3,002	5,000
run time	10.24	18.28	37.13	59.65	115.85	211.58	381.81
depth	10,001	20,000	50,000	100,001	150,002	250,001	
run time	850.11	2,550.95	13,147.40	55,252.44	150,520.09	286,834.95	

Table 6.4. Tabular version of Figure 6.2(d).

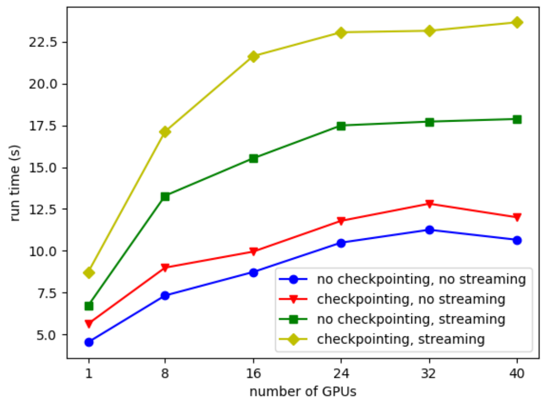
depth	tensor streaming	unified memory no prefetching	unified memory prefetching
152	9.50	8.86	14.03
302	18.73	27.82	42.41
602	38.18	78.66	124.69
1,001	61.49	347.65	449.97
1,502	118.37	1,215.70	1,583.53
3,002	210.89	3,854.67	5,069.40
5,000	377.18	14,061.10	15,550.48

6.2 Semantic Segmentation

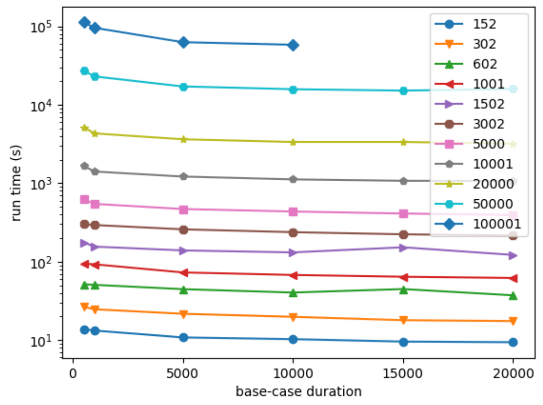
Image classification, as performed by ResNet, is the task of labeling an entire image with a single class. Semantic segmentation is the task of labeling each pixel in an image with a class. The current highest-performing system [57] for doing semantic segmentation on the Microsoft COCO dataset [58] is DRANet [59]. DRANet is based on ResNet, using ResNet as the backbone of an encoder while adding a custom decoder, allowing it to similarly scale to arbitrary depths. The original published version of DRANet had a depth of 101. Here, we reimplement DRANet in Scorch and evaluate variants with depths of 152, 302, 602, 1,001, 1,502, 3,003, 5,000, 10,001, 20,000, 50,000, 100,001, 150,002, and 250,001. For our experiments, we train DRANet on the CityScapes training set [60] for 20 iterations with a batch size of 4 images per GPU. Figure 6.3 repeats the same analyses as Figure 6.2 except that it omits a comparison with PyTorch, as we are unaware of any publicly available implementation of DRANet in PyTorch. Note that Figure 6.3 exhibits the same broad pattern of run times as Figure 6.2, suggesting that the performance enhancements due to divide-and-conquer checkpointing and tensor streaming are robust and apply generally.

6.3 GPT-3

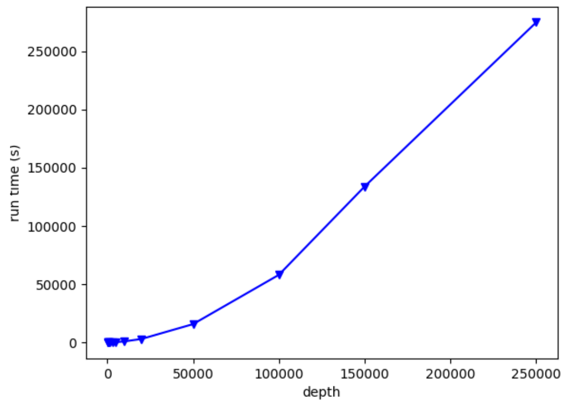
Transformers are a deep-learning architecture receiving significant current attention in the natural-language processing community [61]. The largest transformer with publicly available code and pretrained models is GPT-2 [8]. While the original implementation of GPT-2 is



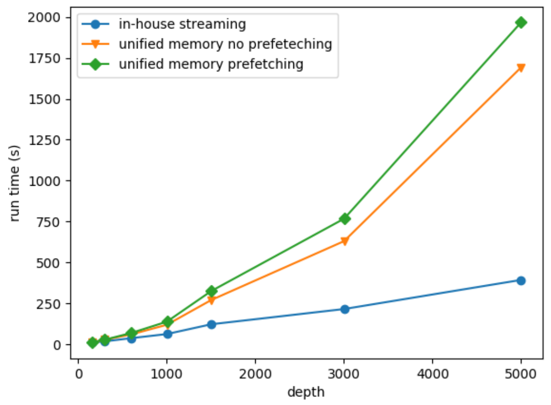
(a)



(b)



(c)



(d)

Figure 6.3. Run times (in seconds) for 20 iterations of DRANets of various depths, with and without divide-and-conquer checkpointing with various base-case-durations, with and without various tensor-streaming methods, and with a batch size of 30 per GPU. There are 8 GPUs per node, so 16 GPUs is over two nodes, 24 GPUs is over three nodes, etc. (a) and (d) use a base-case duration of 20,000. (c) uses a base-case duration of 10,000. A depth of 152, tensor streaming, and a single GPU are used when not specified. Tabular versions in Table 6.5, Table 6.6, Table 6.7, and Table 6.8.

Table 6.5. Tabular version of Figure 6.3(a).

divide-and-conquer	tensor	number of GPUs					
checkpointing?	streaming?	1	8	16	24	32	40
		4.54	7.33	8.73	10.49	11.26	10.66
✓		5.64	9.00	9.95	11.79	12.81	12.00
	✓	6.72	13.29	15.53	17.49	17.72	17.88
✓	✓	8.72	17.13	21.63	23.06	23.14	23.65

Table 6.6. Tabular version of Figure 6.3(b).

	base-case duration					
depth	20,000	15,000	10,000	5,000	1,000	500
152	9.39	9.57	10.28	10.80	13.23	13.77
302	17.47	17.94	19.80	21.63	24.70	26.54
602	37.20	44.72	40.36	44.61	50.59	51.46
1,001	61.91	64.17	67.63	72.84	92.48	94.20
1,502	121.75	152.42	131.15	138.99	155.40	174.64
3,002	214.91	223.00	237.50	258.19	292.94	301.05
5,000	392.81	410.57	435.44	468.61	544.68	620.74
10,001	1,068.42	1,073.65	1,118.32	1,215.36	1,408.04	1,655.18
20,000	3,194.90	3,356.24	3,356.70	3,629.19	4,309.78	5,147.45
50,000	15,897.71	15,146.42	15,781.00	17,125.07	22,952.32	27,140.59
100,001			58,175.23	62,774.44	95,554.15	112,486.25

Table 6.7. Tabular version of Figure 6.3(c).

depth	152	302	602	1,001	1,502	3,002	5,000
run time	9.39	17.47	37.20	61.91	121.75	214.91	392.81
depth	10,001	20,000	50,000	100,001	150,002	250,001	
run time	1,068.42	3,194.90	15,897.71	58,175.23	133,972.58	274,792.20	

Table 6.8. Tabular version of Figure 6.3(d).

depth	tensor streaming	unified memory no prefetching	unified memory prefetching
152	9.39	10.80	11.74
302	17.47	27.40	27.79
602	37.20	60.07	68.97
1,001	61.91	118.84	137.62
1,502	121.75	269.95	324.73
3,002	214.91	629.95	767.03
5,000	392.81	1,691.66	1,966.71

in TensorFlow [62], a version rewritten in PyTorch is available [63]. The developers of GPT-2 have enlarged and enhanced it to yield GPT-3 [9]. GPT-3 is currently the largest published transformer model. However, neither the code nor pretrained models are publicly available.

The architecture of GPT-2 and GPT-3 are identical, differing only in the choice of various architectural hyperparameters. Like ResNet, the GPT architecture is formulated as the repetition of a variety of blocks; changing the amount of repetition can lead to architectures of different depths. The paper that introduced GPT-3 evaluated a number of intermediate architectures with depths between that of GPT-2 and that of GPT-3. GPT-3 is the largest known neural-network architecture. With 175 billion parameters, it can only be trained on a large cluster of GPUs.

> All models were trained on V100 GPUs on part of a high-bandwidth cluster provided by Microsoft [9, p. 9].

It appears that this cluster has 10,000 GPUs.

> OpenAI trains all of their AI models on the cuDNN-accelerated PyTorch deep learning framework. Earlier this month Microsoft and OpenAI announced a new GPU-accelerated supercomputer built exclusively for the organization. “The supercomputer developed for OpenAI is a single system with more than 285,000 CPU cores, 10,000 GPUs and 400 gigabits per second of network connectivity for each GPU server,” the companies stated in a blog. [64], [65]

We have reimplemented the GPT architecture in Scorch. The essence of our implementation is shown in Figure 6.4. All of the variants of GPT-3 from the original paper can be formulated by changing the four hyperparameters `d-model`, `dv`, `n-heads`, and `n-decoders`. The training set for GPT-3 is not publicly available. For our experiments, we train all of the variants of GPT-3 from the original paper for 10 iterations on the WMT’14 English-German dataset [66]. Run times are presented in Figure 6.5. All runs use s single GPU, with divide-and-conquer checkpointing and tensor streaming, for 10 iterations, with a batch size of 1,000. The larger models do not run with larger base-case durations because the

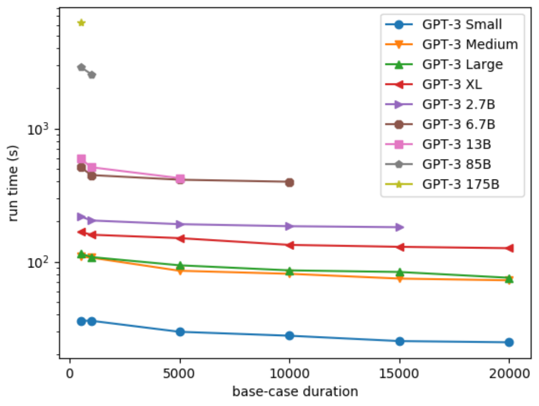
Table 6.9. Tabular version of Figure 6.5.

model name	number of parameters	base-case duration 20,000	15,000	10,000	5,000	1,000	500
GPT-3 Small	125.0M	24.77	25.31	27.76	29.72	35.94	36.07
GPT-3 Medium	356.0M	72.35	74.57	80.98	85.39	107.05	109.29
GPT-3 Large	760.0M	75.59	83.57	85.96	93.79	108.11	114.81
GPT-3 XL	1.3B	126.08	129.11	133.49	150.08	159.13	167.21
GPT-3 2.7B	2.7B		181.20	184.39	191.04	203.65	218.10
GPT-3 6.7B	6.7B			397.82	412.05	446.24	510.59
GPT-3 13B	12.9B				423.59	510.60	592.15
GPT-3 85B	85.0B					2,549.23	2,893.31
GPT-3 175B	174.6B						6,213.83

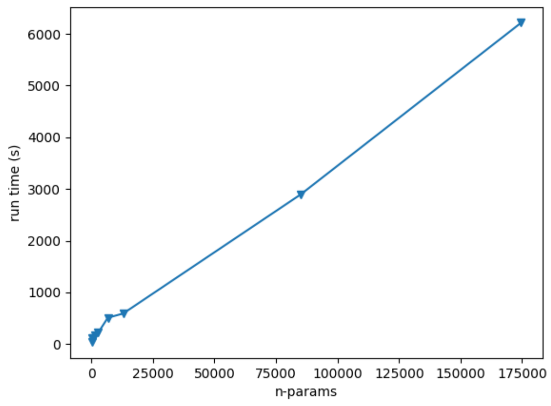
sizes of the individual layers change in addition to the depth of the model, precluding fitting the larger models in GPU RAM with larger base-case durations. Here again, Figure 6.5(a) shows that our implementation incurs little overhead in the interruption and resumption mechanism. Figure 6.5(b) shows that our implementation scales slightly superlinearly with the number of parameters, which is related to network depth.

```scheme
(define (((masked-self-attention dv index) (list Wq Wk Wv)) x)
 (let* ((Q (addmm x Wq))
        (K (addmm x Wk))
        (V (addmm x Wv))
        (QK (addmm Q (transpose K)))
        (QK-div (/ QK (sqrt dv)))
        (QK-div-masked (mask QK-div index))
        (soft-QK-div (softmax QK-div-masked)))
  (addmm soft-QK-div V)))

(define (((multi-head-attention dv n-heads index) (list attention-weights Wo)) x)
 (let loop ((Z '()) (n-heads n-heads) (attention-weights attention-weights))
  (if (= n-heads 0) (addmm (concat (reverse Z)) Wo)
      (loop (cons (((masked-self-attention dv index) (first attention-weights)) x) Z)
            (- n-heads 1)
            (rest attention-weights)))))

(define (((decoder dv n-heads index)
          (list attention-weights fc1-weights fc2-weights bn1-weights bn2-weights)) x)
 (let* ((attention-output
         (layer-normalization
          (+ x (((multi-head-attention dv n-heads index) attention-weights) x))
          bn1-weights))
        (f1-out (((fc-layer) fc1-weights) attention-output))
        (f2-out (((fc-layer) fc2-weights) (((GeLU-layer)) f1-out))))
  (layer-normalization (+ attention-output f2-out) bn2-weights)))

(define (((decoder-stack dv n-heads n-decoders index) weights) x)
 (let loop ((x x) (weights weights) (n n-decoders))
  (if (= n 0)
      x
      (loop (((decoder dv n-heads index) (first weights)) x)
            (rest weights)
            (- n 1)))))

(define (((gpt d-model dv n-heads n-decoders n-timesteps)
          (list Iembed Oembed decoder-weights)) x)
 (let* ((x-embed (addmm x Iembed))
        (decoder-output
         (let loop ((x x-embed) (n 0))
          (if (= n n-timesteps)
              x
              (loop (((decoder-stack dv n-heads n-decoders n) decoder-weights) x)
                    (+ n 1))))))
  (addmm decoder-output Oembed)))
```

Figure 6.4. The essence of our Scorch implementation of GPT.



(a)



(b)

Figure 6.5. Two graphs depicting the run times of the GPT example. The first graph depicts the run time vs. the base-case duration for GPT with various numbers of parameters. It shows that varying base-case duration does not hurt performance that much. The second graph depicts run time vs. number of GPT parameters, increasing to very large numbers of parameters. It shows that run time scales linearly with number of parameters.

7. RELATED WORK

The major novelty of our framework is having support for both divide-and-conquer checkpointing and tensor streaming in an integrated fashion. Let us elaborate on why this is important. The combination allows running applications that are astronomical in size i.e., ResNet-250k, DRANet-250k, and GPT-3, on a single GPU. This requires both the above features and cannot be accomplished with either one of them alone.

There are two components to a deep-learning program that require memory: the weights and the activations of the intermediate layers, which are collectively called the tape, needed to run the reverse sweep. The tape is ephemeral; it can be recomputed and is typically much larger than the weights, which cannot be recomputed. Table 3.1 compares the sizes of the weights and the tape for ResNets of various depths. Note that for ResNets deeper than 152 layers, the tape will not fit in a 12 GB GPU RAM. For ResNets deeper than 10,000 layers, even the weights will not fit in a 12 GB GPU RAM. Divide-and-conquer checkpointing and tensor streaming are complementary. Divide-and-conquer checkpointing reduces the memory footprint of the tape by recomputing activations instead of keeping them on a tape. Tensor streaming reduces the memory footprint of the weights by storing them on the CPU and copying them to the GPU only when required. Scorch affords the user control over which tensors are streamed by providing two kinds of tensors: GPU tensors that are not copied back and forth between the CPU and GPU and streaming tensors that are. We use GPU tensors for storing the tape, thus never copying the tape. We use streaming tensors only for storing the weights.

Our main objective here is to hide the transport time of the tensors under the GPU computation so that the GPU doesn't stall waiting for the data to become available. Since weights are generally smaller than activations, the copying cost for weights can be hidden under the computation cost, if performed in parallel; the copying cost for activations most often cannot. Table 3.3 shows the execution time of a kernel, the time it will take to transport the activation tensor, and the time it will take to transport the weight tensor for some of the most frequently used convolutional-layer sizes in the ResNet network. We can see that the activation transport time is greater than the execution time, but the weight transport time is less than the execution time. From Table 3.1 we can also see that the tape of ResNet-50k needs 2.8TB of RAM, ResNet-100k needs 5.7TB, and ResNet-1M needs 57TB of RAM. Even the most modern computers may not be equipped with so much RAM and it won't be possible to run these programs if one were to offload the tape to the CPU. But with divide-and-conquer checkpointing, the need to store the tape is significantly reduced, and ResNets of such sizes can be run on current computers.

Existing work that tries offloading the tape to the CPU realizes this bottleneck and tries to offload only a part of the tape to the CPU. This has two problems. First, this puts a limit on how much of the tape can be offloaded before it starts affecting performance, inhibiting these approaches from scaling to astronomical sizes. Second, this creates a need for a scheduling algorithm to determine which parts of the tape should be offloaded when, for optimal performance.

7.1 Checkpointing

Volin and Ostrovskii [32] presented a method for individual checkpointing (Figure **??**b) at function-call boundaries. Many systems perform individual checkpointing hardwired in specific neural-network layer types that are recomputed instead of having their outputs stored on the tape. Bulo, Porzi, and Kontschieder [67] did this for batch-normalization layers. Individual checkpoints alone do not yield sublinear space penalty. Gruslys, Munos, Danihelka, *et al.* [33] presented a method for left-branching checkpointing (Figure **??**c). Chen, Xu, Zhang, *et al.* [34] and Pudipeddi, Mesmakhosroshahi, Xi, *et al.* [68] presented methods for right-branching checkpointing (Figure **??**d). Chen, Xu, Zhang, *et al.* [34] used this to achieve square-root checkpointing. Griewank [10], Gruslys, Munos, Danihelka, *et al.* [33] and Feng and Huang [69] presented methods for divide-and-conquer checkpointing (Figure **??**e,f). This work applied in the abstract to a sequence of program steps or neural network layers and lacked the ability afforded by our interruption and resumption mechanism to apply to arbitrary differentiable programs. Griewank [10], Gruslys, Munos, Danihelka, *et al.* [33] and Feng and Huang [69] also presented methods for scheduling checkpoint intervals. This work is orthogonal to our work presented here. Scorch implements all of the scheduling methods of Griewank [10] and could presumably be extended to also implement all of the scheduling methods of Gruslys, Munos, Danihelka, *et al.* [33] and Feng and Huang [69]. Kukreja, Hückelheim, and Gorman [70] presented a method for offloading checkpoints to the CPU RAM or another memory bank to allow a larger number of checkpoints. Our results suggest that the limited PCIe-bus bandwidth should be used to stream the weights so that the GPU does not stall waiting for data rather than using it to stream checkpoints, which are activations, are are typically much larger than the weights.

TAPENADE [47] provided mechanisms for manual static annotation of individual checkpoints in the program text (for function bodies, call sites, and arbitrary code blocks) both through pragmas and command-line arguments to the preprocessor. It also implemented divide-and-conquer checkpointing, but only when the checkpoint intervals corresponded to individual non-nested manually-annotated FORTRAN `DO` loops. ADOL-C [71] provided a nested taping mechanism for the special case of time-integration processes [72] that also implemented divide-and-conquer checkpointing, but only for this special case. Vieira [73] provided an implementation of Gruslys, Munos, Danihelka, *et al.* [33] that was limited to the case of backpropagation through time (BPTT) in recurrent neural networks (RNNs). PYTORCH provides support for manual annotation of individual checkpoints via `torch.utils.checkpoint.checkpoint` and manual annotation of right-branch checkpointing via `torch.utils.checkpoint.checkpoint_sequential` [74]–[76]. The default parameter settings for the latter do not yield square-root checkpointing and sublinear space penalty. Further, the latter only supports networks structured as a linear chain and thus does not apply to networks that contain residual connections (as do ResNet and DRANet) or other more general graphs such as GPT. The official TENSORFLOW release lacks support for checkpointing. A third-party enhancement [77] provides support for right-branching checkpointing but only for TENSORFLOW 1. It can achieve square-root checkpointing through use of a heuristic in some cases. Another third-party enhancement [78] provides support for manual annotation of individual checkpoints around functions in TENSORFLOW 2. All of the above requires manual annotation of checkpoint intervals that could only be placed around specific kinds of constituents in specific kinds of programs or neural networks. This often does not yield sublinear space penalty. SCORCH requires no manual annotation from the user to perform divide-and-conquer checkpointing and does not impose any constraint on program structure. Just by changing the reverse AD invocation of `*j` with `checkpoint-*j`, balanced divide-and-conquer checkpointing is automatically performed on the entire computation, yielding sublinear space penalty for any computation.

7.2 Tensor Streaming

Prior work has posed tensor migration scheduling as an optimization problem and has developed various approaches to solving it. Zhang, Yeung, Shu, *et al.* [79] presented a heuristic based on tensor size and the duration of a program interval where that tensor is not accessed to decide which tensors to offload and when. Wang, Ye, Zhao, *et al.* [80] presented a different heuristic, offloading the activations of convolutional layers as they are computed, but not other layers, to the CPU, to fit the GPU RAM budget specified by the user. Hildebrand, Khan, Trika, *et al.* [81] used two different CPU memory pools, each of which had different characteristics, and used integer linear programming to decide which parts of the tape to offload to which of the CPU memory pools. Huang, Jin, and Li [82] used a genetic algorithm to optimally reorder the sequence of operations to get a better alignment with offloading the tape to the CPU. Rhu, Gimelshein, Clemons, *et al.* [83], Meng, Sun, Yang, *et al.* [84], Jin, Liu, Jiang, *et al.* [85], Chen, Chen, and Hu [86], Le, Imai, Negishi, *et al.* [87], and Peng, Shi, Dai, *et al.* [88], used the time between reuse of a tensor to decide which tensors to offload and when. However, the methods of Rhu, Gimelshein, Clemons, *et al.* [83] and Chen, Chen, and Hu [86] only applied to a primal that could be expressed as a simple sequence of layers. Ren, Luo, Wu, *et al.* [89] presented migration strategies based on the sizes and access frequencies of tensors. Ren, Rajbhandari, Aminabadi, *et al.* [90] extended this work to offload some of the computation to the CPU as well. Guo, Liu, Wang, *et al.* [91] posed migration scheduling as a constraint-satisfaction problem (CSP). Solving this CSP yielded a solution to achieve the desired FLOPs while maintaining a given GPU RAM budget under the PCIe bandwidth constraints. Jhu, Liu, and Wu [92] used dynamic programming while Sekiyama, Imamichi, Imai, *et al.* [93] used mixed integer programming to schedule migration. All this work is only able to offload a part of the tape, not the entire tape, to the

CPU, because offloading the entire tape would degrade performance. Scorch never offloads the tape to the CPU because divide-and-conquer checkpointing allows it to be recomputed, which is faster than offloading and refetching it from the CPU. Because Scorch never offloads the tape to the CPU, it doesn't need to solve the associated migration scheduling problem.

Our tensor-streaming mechanism has three benefits. Since Scorch is a functional language, we are guaranteed no mutation, which means that, once used, the tensors copied to the GPU can be evicted and don't need to be copied back to the CPU, so long as the original copy is kept on the CPU. This saves us half the communication time incurred by other duplex streaming systems. Since tensor streaming is implemented at the language-implementation level, it can do both pre-planned fetching, by using `get-streaming-plan` to create a streaming plan in advance, and demand fetching in programs with control flow where a pre-determined order cannot be obtained without running the computation.

Shirahata, Tomita, and Ike [94] reused the tensors on the tape that store activations from the forward sweep to store activation gradients during the reverse sweep. Chen, Xu, Zhang, *et al.* [34], Wang, Ye, Zhao, *et al.* [80], Jin, Liu, Jiang, *et al.* [85], and Zhang, Yeung, Shu, *et al.* [79] performed liveness analysis to determine which tensors to free or to reuse. Scorch does not need to perform liveness analysis since it is a garbage collected (GC) language. Further, the reuse achieved by Shirahata, Tomita, and Ike [94] comes for free with GC. Much like Rhu, Gimelshein, Clemons, *et al.* [83] and Wang, Ye, Zhao, *et al.* [80], we implement our own memory-pool manager and do not use `cudaMalloc` and `cudaFree` for allocating individual tensors, because calls to these functions are synchronous and incur non-negligible overhead. Finally, practically all of the techniques discussed in early work on GPU memory management [95] are now incorporated into CUDA and most other frameworks.

7.3 Combinations of Checkpointing and Tensor Streaming

Pudipeddi, Mesmakhosroshahi, Xi, *et al.* [68] combined right-branching checkpointing with methods for offloading the checkpoints to the CPU, and streaming the weight tensors to the GPU from the CPU. However, their approach was limited to the highly specific case of BERT models and the checkpoint intervals always begin and end at the end-points of an encoder or decoder model. Moreover, the PCIe-bus bandwidth was shared between transporting the checkpoints and the weights. The weights were stored on the CPU and the update step was also performed on the CPU. The SCORCH `*j-update` primitive fuses the gradient computation with the update step which allows the update step to be to performed on the GPU as soon as the gradients are calculated. Their system was only able to run models with 50 billion parameters. We suspect that CPU RAM was the limiting factor, since it was used for both the weights and checkpointed activations. SCORCH on the other hand can run GPT-3 with 175 billion parameters on a machine with 768TB of CPU RAM, the limiting factor being the CPU RAM used only by the weights. Tezikov [96] provided an implementation of Pudipeddi, Mesmakhosroshahi, Xi, *et al.* [68] that applied more generally to a simple sequence of neural-network layers, but not to arbitrary neural networks or arbitrary differentiable programs.

7.4 Various Alternative Training Strategies

The goal of this project was to provide the ability to instantiate and train extremely large scale neural networks on a single GPU of small size. We made that possible through the use of divide-and-conquer checkpointing and weight streaming. There are other alternative training strategies available that would allow training of such large neural networks on a single GPU without the use of checkpointing.

Jain, Phanishayee, Mars, *et al.* [97] compressed the taped output of certain neural-network layers in a lossy fashion during the forward sweep and decompressed them during the reverse sweep. It is incomparable to our lossless exact method. Gomez, Ren, Urtasun, *et al.* [98] presented a reversible variant of ResNet where the input activation could be approximately reconstructed from the output activation during the reverse sweep, eliminating the need to store a tape. This work only applies to the special case of (approximately) invertible program steps, unlike our work that applies when this does not hold.

Another line of work [99], [100] shows how to use second order methods (Newton's method) to train networks instead of using first order methods. The second order methods are generally expensive as they require the computation of inverse Hessians, however these works show how to effectively sample the Hessians to approximate the curavature and carry out optimization that is faster and has better statistical properties than the first order method. Note that these methods are a bid to reduce the run time of the program, not the memory foot print. These methods can be used in conjunction with the checkpointing in Scorch to create much bigger networks and train using second order methods with divide-and-conquer checkpointing.

Yuan, Wolfe, Dun, *et al.* [101] inspired by the dropout technique, provide a method in which a network is split into subnetworks and each network is trained independently and then combined to yield an aggregated performance similar to that of training the entire network together. They provide convergence guarantees for their algorithm. This is indeed a rivaling technique to checkpointing, however we point out some key limitations when compared with Scorch. Scorch allows training a deep neural network on a single GPU with logarithmic time overhead incurred due to checkpointing. If we assume a 12GB GPU and assume that the biggest ResNet trainable on this GPU without checkpointing is 152 layers, then to train a ResNet with 302 layers with checkpointing will incur log(2) overhead. Using the method of Yuan, Wolfe, Dun, *et al.* [101], if one was to split ResNet 302 into two subnetworks, it would take twice as long since each of those has to be trained independently. Our method scales logarithmically whereas [101] scales linearly. Secondly, separate training of each subnetwork and then concatenating the network does not equate to training a combined whole network. Several splits would need to be made rather than just 2 so that some parts of the model can be overlapped while training in order to arrive at the desired performance level. Therefore the scaling is likely worse than linear.

7.5 Benchmarking Against Related Work

Of all of the prior work on checkpointing and tensor migration discussed in Section 7, in addition to the implementations of checkpointing included in PyTorch, we were only able to find public code for Superneurons [80], DeepSpeed [90] (an updated version of Sentinel [89]), the implementation of l2l [68] provided by Tezikov [96], moDNN [86], vDNN

[83], the implementation of Gruslys, Munos, Danihelka, *et al.* [33] provided by Vieira [73], and the two third-party implementations of checkpointing for TensorFlow [77], [78]. For PyTorch, we only considered the implementation of right-branching checkpointing provided by `torch.utils.checkpoint.checkpoint_sequential` as limited to simple sequences of neural-network layers. Superneurons, moDNN, and vDNN are particular to neural networks and are not capable of running arbitrary differentiable programs. We were not able to compile Superneurons. We were not able to run moDNN and vDNN on any of our examples because they lack implementation of batch normalization that is needed for ResNet and DRANet, and layer normalization and GeLU that are needed for GPT. DeepSpeed and Ł2L only apply to simple sequences of neural-network layers, not arbitrary differentiable programs. We did not consider Vieira [73] as it only applies to backpropagation through time, not general neural networks, let alone arbitrary differentiable programs. We also did not consider AI [77] as TensorFlow 1 is deprecated. We thus compared Scorch with DeepSpeed, which provides tensor migration, but not checkpointing, l2l, which provides tensor migration and right-branching checkpointing, but not divide-and-conquer checkpointing, PyTorch, which does not provide tensor migration but does provide right-branching checkpointing for simple sequences of neural-network layers, and TensorFlow, which does not provide tensor migration, but does provide individual checkpointing.

We ran variants of our ResNet and GPT examples in TensorFlow, PyTorch, DeepSpeed, l2l, and Scorch. For each, we determined the largest variant that can run on a single Titan V GPU (12 GB RAM) connected to a CPU with 768 GB RAM. For ResNet and GPT, we measured the depth, in number of layers, of the largest network that can be trained, i.e., the largest network whose gradient can be computed with reverse AD. The Scorch results employ divide-and-conquer checkpointing on all examples and tensor streaming on the ResNet and GPT examples. The results are shown in Table 7.1. Scorch is able to handle examples at least an order of magnitude larger than these other systems, often far larger than that.

Table 7.1. The maximum network depth, for ResNet and GPT, trainable with SCORCH and various other frameworks, on a single Titan V GPU (12 GB RAM) connected to a CPU with 768 GB RAM.

	ResNet	GPT
TENSORFLOW (individual checkpointing)	608	4
PYTORCH (right-branching checkpointing)	760	48
DEEPSPEED	760	48
L2L	38,000	3,000
Scorch	**250,000**	**14,000**

8. OBJECT CLASSIFICATION FROM RANDOMIZED EEG TRIALS

The processing of visual stimuli in the human brain has been a topic of interest among neuroscientists and machine learning researchers alike. For machine learning researchers, decoding this process could provide vital cues for design and operation of computational models, specifically neural networks. While the overall objective might be beyond the capability of current available technology, smaller efforts to scratch the surface of this subject have been commenced since deep learning came into the main stream almost a decade ago.

The electroencephalography or EEG cap measures the electrical activity on the scalp resultant from underlying brain activity. The cap consists of various electrodes covering different regions of the head and measure the electrical signal on the surface of the head at a high frequency.

The idea of this project is to present a subject wearing an EEG cap with visual stimuli and measure the EEG signal, and then to try to determine what visual stimuli was being presented to the subject from the EEG signal. Consider a computational model that could attain 100% accuracy on this task. We would be able to claim that the computational model is equivalent to the brain's visual stem. For this specific project, the problem was posed as an image classification problem. The subject was shown images belonging to various object categories and instead of trying to determine the actual image, just the image category was tried to be decoded.

In this project, we collected the largest ever single subject EEG data for visual object classification. This dataset was at the bounds of feasibility and has been publically released alongwith the evaluation results of the current state of the art models on it. The results show that the problem is hard and far from unsolved and requires extensive research to make progressive strides.

This project was a combined effort with other students of the lab. The author's contributions to this was the entire data collection as well as implementation of the neural network architectures used to conduct the baseline study on this dataset.

8.1 Motivation

The motivation for this data collection effort came from earlier work in our lab[102] that I will briefly summarize here. A recent CVPR paper[103] claimed to have solved this problem. They collected a data set from 6 subjects. Each subject was shown 50 images each from 40 Imagenet classes. Each image was shown for 0.5 seconds. Images from one class were shown together with no blanking in between images. A 10 seconds blank was inserted between classes. They constructed training and test sets in which samples from all blocks were considered equally for the training and test sets. They achieved 84% classification accuracy on this task using an LSTM based classification network.

We hypothesized that their high accuracy was due to the fact that they showed images from a single class together, thus confounding the clock with the class label. Thus their classifier was classifying the block number instead of the actual class. To test this hypothesis, we constructed our own splits of their data in which we held back data from one subject for testing and trained on just the other five. The classification accuracy fell to almost chance levels. Since this added the additional difficulty of cross subject decoding and could not be used as an evidence to refute the original authors' work, we decided to collect our own dataset.

We used the exact same stimuli and the presentation pattern as used by the original authors but we used both block design and rapid event design. In block design, we showed all images of a class together with blanking in between. For rapid event design, we mixed the images of different classes together. When we trained the classification networks on this dataset, the block design method gave near the reported accuracy from the original paper but the rapid event dataset gave almost chance accuracy.

This conclusively showed us that this problem is hard and unsolved. The data collection protocol of the original authors was not conducive to generating a decent enough dataset to allow solving this problem. We hypothesized that more data might be able to help. Hence we decided to collect our own dataset for the problem.

8.2 Data collection protocol

We selected the same 40 image classes from Imagenet as used by [103]. We selected 1000 images per class instead of 50 images in [103]. Each image was resized to be 1920 × 1080. The images were divided into 100 blocks. Each block contained 400 images and exactly 10 of each class. There were a total of 40000 images.

A single male subject viewed the stimuli over 10 sessions. In each session, 10 blocks were shown. Each image was shown for 2s with 1s of blanking in between. Each block run started with 10s of blanking and ended with 10s of blanking so a block in total took 20 minutes and 20. Each session lasted nominally 6 hours or so and thus the entire data collection effort took almost 60 hours.

The EEG cap had 96 electrodes collecting measurements at 4096 Hz. This data was downsampled to 1024 Hz and only the first 500ms of data was used even though 2 seconds were recorded. Unlike fMRI data where an anatomical brain scan is used to align data from different subjects and across different sessions with the same subject, no such alignment method exists for EEG so best efforts were made to mitigate such misalignments. A single capping was used within a session. The same cap with a pre-cut holes was used across sessions and some of the electrodes were visually made sure to be in the same spot at every capping.

The data was band pass filtered from 1-40 Hz range to remove the DC component as well as high frequency noise. The data was then partitioned into five equal sized splits, each split containing the same number of samples from each class.

As a result of this effort, we collected the largest ever single subject EEG visual stimuli dataset for classification. This effort took about 6 hours per session and 10 sessions in total which is almost two business weeks. Fewer labs or subjects would be willing to go through the hardship of collecting such a dataset.

8.3 Analysis

We conducted some preliminary analysis on this dataset to determine how well the current state of the art methods can solve the problem of classification from EEG data. We used

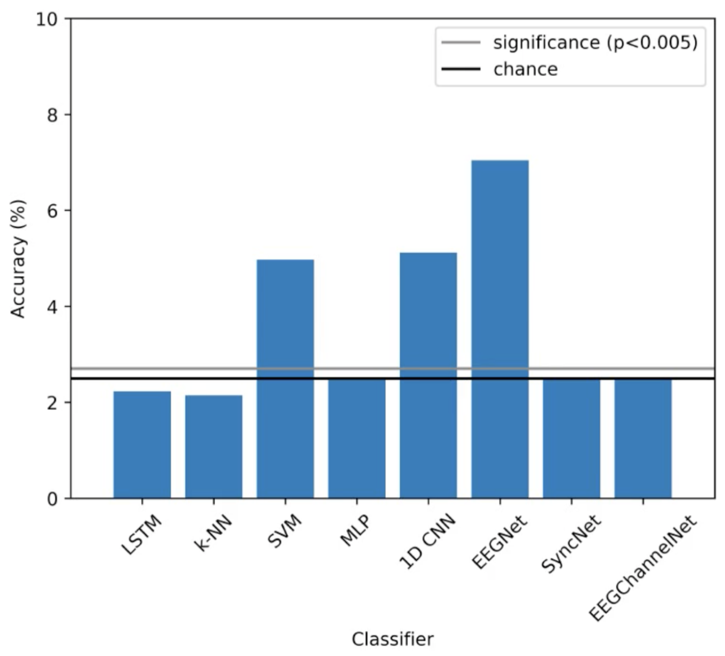


Figure 8.1. Classification accuracies of various classifiers on the collected dataset

8 classifiers: LSTM, k-NN, SVM, a two layer fully connected network (MLP), a 1D CNN, EEGNet[104], SyncNet[105] and EEGChannelNet[106].

8.3.1 Maximum achievable accuracy

We first investigated whether the task is possible with the current state of the art methods. We trained all the 8 classifiers on the data. The classification accuracies are shown in Fig. 8.1. Only three classifiers, SVM, 1D-CNN and EEGNet were able to get statistically significant above chance accuracies. This indicates that a lot of algorithm improvements need to be made to make progress on this task.

8.3.2 Amount of data required

We investigated how much data is required to be able to train the classifiers to the accuracy that we report in the above section. We trained on increments of 10% of data. The results are shown in Fig. 8.2. We concluded that about 60% of the total data collected is sufficient to train the classifiers to the maximum accuracy.

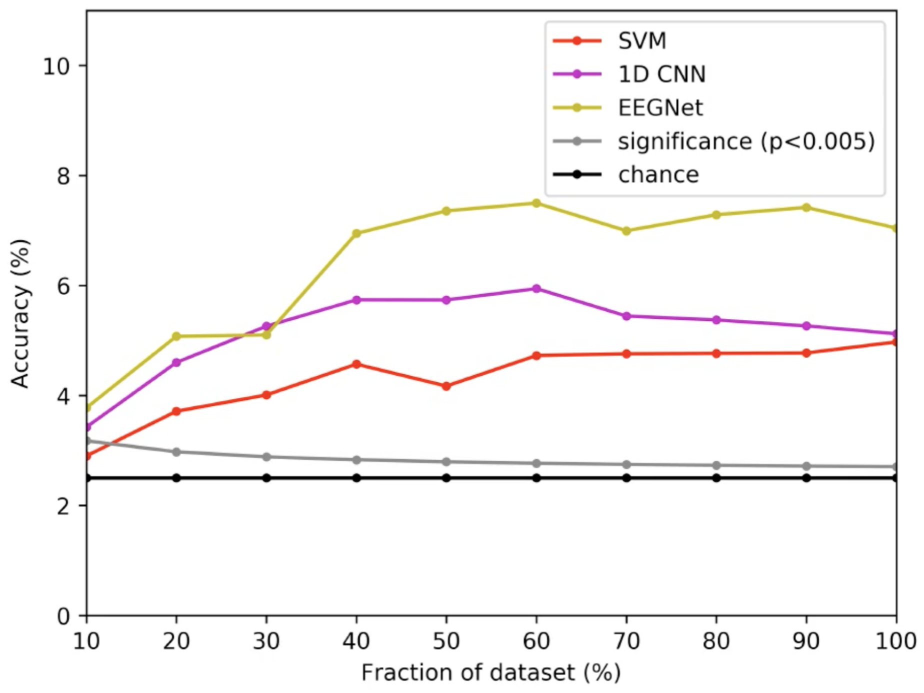


Figure 8.2. Classification accuracies as a function of the fraction of dataset

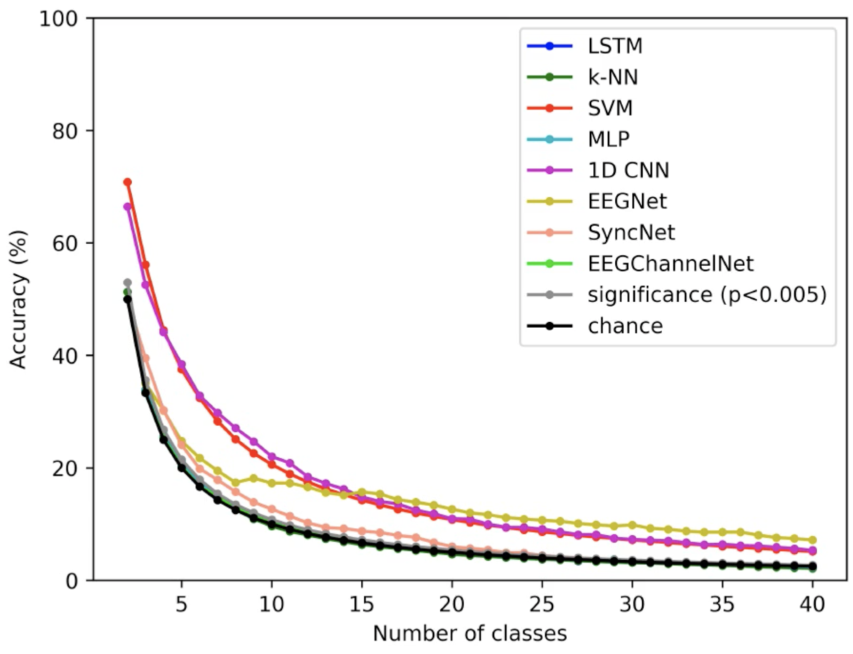


Figure 8.3. Classification accuracies as a function of classes

8.3.3 Maximum number of decodable classes

We investigated how many classes are decodable with the current state of the art classifiers. To do this, we conducted a greedy analysis by training on sets of 2, 3, 4 all the way to 40 classes. The results are shown in Fig. **??**. We see that the accuracies start to taper off as we get to 40 classes, indicating that more classes would not be decodable and will yield chance performance.

8.4 Summary

In this work, we collected the largest single subject EEG dataset for object classification and reported the performance of state of the art methods on this dataset. The results indicate that the problem is hard and unsolved and requires significant research efforts to make progress.

9. SPATIO-TEMPORAL ACTIVITY LOCALIZATION IN UNTRIMMED UNCROPPED VIDEOS

Much of the previous work on activity recognition in video has focused primarily on the following tasks:

- Activity classification: Given a temporally trimmed and spatially cropped video clip, classify the activity occurring in it. Kinetics[107] and UCF101[108] are examples of such action classification datasets.

- Temporal activity localization: Given a temporally untrimmed but spatially cropped clip, detect the temporal bounds of an activity within the clip as well as classify the activity [109], [110].

- Spatial activity localization: Given a temporally trimmed but spatially uncropped clip, detect the spatial region of an activity within the clip as well as classify the activity [111].

A combined problem in which the activity is neither spatially, nor temporally cropped around the action is harder especially when applied to surveillance videos because the incoming video stream is continuous and actions seem to take place in small regions and for short duration. Applying this to surveillance cameras also imposes constraints on compute power since there could be many surveillance cameras in a given space and only a few compute nodes equipped with sufficient compute power available to handle them.

In this chapter, I present our work on spatio-temporal activity localization in videos that were both untrimmed in the temporal domain and uncropped in the spatial domain. We created an extremely fast yet accurate method that achieved localization as well as classification of 37 activities on the MEVA (Multiview Extended Video with Activities) dataset while running about 3x real time. We achieved state of the art performance on the MEVA dataset using this method.

Our method consists of three main stages:

1. Temporally splitting a video into clips and then spatially localizing candidates of interest (proposals) within such clips.

2. Filtering proposals and classifying activities.

3. Filtering heuristics tailored to the dataset and evaluation metrics to improve results.

Details of the dataset, the evaluation metrics, the system design and the results is detailed below. The entire project had contributions from other members of the lab as well. The author's contributions were designing and coding the entire pipeline, motion filtering of proposals, training and implementing activity classifiers and implementing the filtering heuristics.

9.1 Dataset

This project was carried out under the IARPA DIVA grant who provided the MEVA dataset for us to devlop and test on. The MEVA dataset was filmed using actors performing pre-defined activities. It contains more than 3,000 five-minute-long videos, captured from surveillance cameras positioned in various indoor and outdoor locations. There were a fixed number of such cameras and each video file comes with the ID of the camera that was used to film it.

The dataset contains 37 activities classes. Each video can contain multiple instances of multiple activities. Activities can also potentially occur simultaneously in multiple regions of the video at the same time. The MEVA dataset was annotated by a collaborative effort among various participants of the IARPA DIVA program as well as Kitware, one of the DIVA system evaluators. These annotations denoted the spatial location of activity instances with bounding boxes and the temporal location of activity instances with start and end times. We used the MEVA videos together with the annotation provided by DIVA participants to develop and train our system. We used the MEVA videos annotated by Kitware to validate our system in-house since the sequestered set used to evaluate the submitted systems was also annotated by Kitware.

9.2 Evaluation Metrics

Since the task is of spatio-temporal localization, the evaluation metrics specified by the IARPA DIVA program were:

- probability of missed detection (P_{miss}),
- time-based false alarm (T_{fa}),
- detection error tradeoff (DET curve), and
- mean, normalized, partial area under the DET curve (mean-nAUDC).

For a ground truth activity to be considered detected, the system must produce a detection that overlaps with it by atleast 1 second. Spatial localization was not a part of these metrics to ease the difficulty for the participants. P_{miss} is the fraction of ground-truth activity instances that were not detected by the system. It ranges from zero to one, with lower being better, and is analogous to one minus recall. T_{fa} represents the fraction of time that the system produced false-positive activity detections. It also ranges from zero to one, with lower being better, and is analogous to one minus precision. For a ground-truth activity instance to be considered detected, a detection of the same class must overlap the ground-truth activity instance temporally by at least 1 s and have a higher confidence score than a specified threshold. This threshold is varied to produce different pairs of P_{miss} and T_{fa} values. This produces a DET curve, which is analogous to a precision-recall curve, and allows for the computation of a mean-nAUDC score for the range of T_{fa} values between zero and a specified upper T_{fa} bound. Mean-nAUDC is analogous to mean average recall. We evaluate our system with the same two metrics that the SDL test server uses to evaluate all submissions: mean-nAUDC in the T_{fa} range $[0, 0.2]$ (nAUDC@$0.2T_{\text{fa}}$) and P_{miss} at a T_{fa} of 0.04 (P_{miss}@$0.04T_{\text{fa}}$). For both metrics, lower scores indicate better performance. Hereafter, we use nAUDC to refer to mean-nAUDC@$0.2T_{\text{fa}}$ and P_{miss} to refer to P_{miss}@$0.04T_{\text{fa}}$.

9.3 System Design

The complete design of our system pipeline is shown in Fig. 9.1.

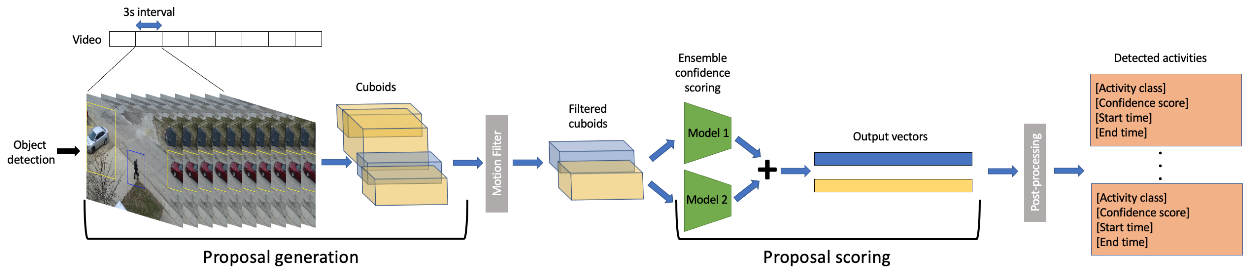


Figure 9.1. System architecture.

9.3.1 Temporal splitting

Given an input video of 5 minutes, we split it into chunks of 3 seconds each and process each chunk as a separate sample. Splitting into smaller chunks works well because the length of an activity is tied to the nature of the activity. For example, door open or door close normally do not take more than a few seconds. Most activities in the DIVA dataset were of the nature that their lengths were smaller than 3 seconds. There were some longer activities like using laptop computer but such activities are stationary for the most part and can be detected across multiple 3 second chunks.

The 3 second split was an overlapping split i.e. we split every second for the next 3 seconds to be able to efficiently capture the start and end times of activities. A temporal non-maximal suppression will be applied at the end to deal with multiple detections of the same activity coming from such overlapping intervals. We did not localize an activity instance within the 3 second interval i.e. 3 second is just about the length we would assign to each activity detection, no more and no less. This did not hurt on the DIVA dataset given their metrics because they allowed for sufficient overhang of detections over ground truth activity instances.

9.3.2 Spatial proposal generation

Once we temporally split the video into smaller 3 second chunks, the next step is to spatially crop out regions of interest that could serve as candidates for activities. We tested several methods in an effort to find out the optimal proposal generation mechanism.

- Since we know the camera id for each video, and given a certain surveillance camera installed in a certain space, activities are likely to occur in more or less the same region. We accumulated ground truth activities from the dataset per camera and constructed heat maps to detect such regions of high interest. We used connected components to create cuboids from such heatmaps and used them to serve as the candidate proposals.

- We tried a brute force overlapping cuboids over the entire field of view.

Table 9.1. Proposal-generation mechanisms, their recall, and average number of proposals per 3 s interval. For Object-Detector Cuboids, the number refers to the expansion factor.

Proposal-generation mechanism	Recall	Average number of proposals per 3 s interval
Nonoverlapping Tiled Cuboids	41%	201.0
Overlapping Tiled Cuboids	79%	707.0
Heatmap Cuboids	21%	4.6
Object-Detector Cuboids 20%	63%	20.0
Object-Detector Cuboids 30%	67%	20.0

- We used a trained off the shelf YOLO [17] object detector to detect objects. We knew that most of the 37 activities of interest in the MEVA dataset are performed by humans of occur around vehicles such as car, truck, bicycle etc. We applied the YOLO detector to the first frame of the 3 second chunk, expanded them by either 20 or 30% to generate such regions and used them as our candidate proposals.

Experiments showed that the best results were obtained by using the YOLO detector expanded by 30%. Results are shown in Table 9.1.

9.4 Motion Filtering

Nominally, one would compute a confidence score for each cuboid produced by our proposal-generation mechanism to output as an activity detection. However, many cuboids do not exhibit any motion and thus likely don't contain any activity instances and will likely be assigned a low confidence score, or even eliminated using heuristics to be described later. Thus, we can filter out these cuboids using motion detection and eliminate the need for assigning them a confidence score. This can speed up overall processing and improve overall evaluation metrics because motion detection can be much faster and much more reliable than using our repurposed activity classifier to produce confidence scores.

Table 9.2. Comparison of different motion thresholds

t_2	nAUDC	P_{miss}
0.00	0.34825	0.49548
0.10	0.34304	0.49252
0.15	0.34865	0.49276
0.20	0.35795	0.49709

We gauge the total amount of motion in a cuboid and discard cuboids with very low or no motion. We use image differencing to detect motion. Given two successive frames, f_1 and f_2, we create a binary, single-channel, difference image

$$f_d = \left[\sqrt{\sum_c \left(\frac{f_2 - f_1}{255.0} \right)^2} > t_1 \right] \tag{9.1}$$

where $\sum\limits_c$ sums over the channel dimension of the frame, t_1 is a per-pixel motion threshold, and $[\cdot]$ is 1 if $\cdot$ is true and 0 otherwise. To compute the motion score p of a proposal whose spatial extent is (x_1, y_1, x_2, y_2), we sum the values inside the proposal box in the difference image, normalized by the area of the proposal.

$$p = \frac{\sum\limits_{x=x_1}^{x_2} \sum\limits_{y=y_1}^{y_2} f_d^{(x,y)}}{(x_2 - x_1) \times (y_2 - y_1)} \tag{9.2}$$

We discard proposals with $p < t_2$, where t_2 is a total motion threshold. We use integral images to speed up the computation of (9.2). We set t_1 to 0.1 and evaluated various values of t_2 (Table 9.2, determining that a t_2 of 0.1 worked best.

9.5 Activity Classifier

We used an off the shelf TSM activity classifier for this stage. To generate the training data, we first augment our activity-class set with a *background* class to account for proposals where none of the target activity classes are occurring. We first generated a training set of cuboids, using the methods described above, on MEVA videos. In order to label each cuboid in this training set with an activity class or background, we matched the cuboids to ground-truth annotations based on spatial Intersection-over-Union (IoU) and temporal overlap. For each cuboid, we found all ground-truth activity instances that overlapped the cuboid by $\geq$45 frames. For each of the ground-truth activity instances, we only considered the temporal interval that overlapped the cuboid. Each ground-truth activity instance has bounding boxes for each participant in each frame. We computed a superbox from all of these bounding boxes from all frames that overlapped the cuboid. We then computed the spatial IoU between this superbox and the cuboid box. If this IoU exceeded 0.5, we added the activity label from the ground-truth instance as an activity label for this cuboid. Although rare, it was possible for a cuboid to end up with multiple distinct activity labels. In this case, one training sample was created for each distinct label. If a cuboid was not matched with any annotations, we labeled it as the background class. An overwhelming majority of our cuboids were labeled as background; in an effort to balance our dataset during training, we randomly chose a small subset of these cuboids to be part of our training set. The model was initialized with available pretrained weights [112] on the Kinetics dataset. We then fine-tuned it with weighted cross-entropy loss where the weights corresponded to the inverse of the frequency of the activity class in our training set of cuboids.

Once such a classifier is trained, we can use the softmax score produced by it as actual confidence scores for the activity classes. We trained two models with different proposal generation mechanisms. The first was trained using the method described above. The second was trained by a different temporal splitting strategy. Instead of splitting the videos into 3s chunks, the proposals actually came from temporal ground truth annotations and were resampled to fit the 3s length criteria. We then explored two different approaches for combining or ensembling the output of these two models. The first approach was to simply sums the output scores of each model. The second approach selectively outputs from each model on a per-class basis. For this, we ran our system with two models individually and determined which model performed better on a per-class basis, using the per-class nAUDC and P_{miss}. Then, in the ensemble, the final output vector was formed by using the class scores from the model that had higher performance on that class. The results showed that both methods of ensembling increased performance, with the score-summing approach being slightly better. We use the score-summing ensemble approach in our system based on this analysis.

9.6 Filtering Heuristics

To convert the above activity classifier from a classifier to an activity scorer, we output all the activity detections produced by the classifier instead of just the one with the maximum score. This naturally ends up with tons of detections so we tested various strategies to mitigate this. Our filtering heuristics were designed to encode prior knowledge of which kinds of activities can occur where. We refer to these as *possible activity classes.* We evaluated four different strategies for filtering the set of detections produced:

1. Do not output detections for a cuboid with lower confidence score than the confidence score of the background class for that cuboid.
2. Do not output detections for a cuboid with lower confidence score than the confidence score for any impossible activity class for that cuboid.
3. If the top-scoring activity class for a cuboid is the background class, do not output any detections for that cuboid.
4. If the top-scoring activity class for a cuboid is impossible, do not output any detections for that cuboid.

This is done for every cuboid in each 3 s interval. The total set of detections for a given video is the combination of detections of all non-overlapping 3 s intervals in the video. A consequence of each of the above filtering strategies is that we only output detections for possible activity classes. We ran our system on our validation set with each strategy and measured nAUDC and $P_{\text{miss}}$and found the third and fourth strategies practically equivalent and employed the fourth strategy in our system.

9.7 Possible Activity Classes

To construct the sets of possible acitivity classes, as part of training, we determined which activity classes occur in each camera ID in the training set to produced a sets of camera-based possible activity classes for each camera ID. Alternatively, we manually specify a set of object-based possible activity classes for each object class to indicate which activity classes can occur in the vicinity of each object class. For example, an instance of the `vehicle_turns_left` activity class is likely to occur in the vicinity of a car detection but not a person detection while an instance of the `person_embraces_person` activity class is likely to occur in the vicinity of a person detection but not a car detection.

We performed an analysis to assess the effectiveness of these two priors for specifying possible activity classes. One basis used the camera-based possible activity classes when the camera ID was known, and the object-based possible activity classes otherwise. The second basis used object-based possible activity classes for every cuboid regardless of whether the camera ID was known. The results showed that the first approach yields the best results, and, as such, we used it in our system.

9.8 Additional Filtering

Our proposal generation mechanism allows for multiple cuboids in a 3 s interval, which can result in multiple detections of the same class for each 3 s interval. We need to handle these proposals carefully to reduce the false alarm rate of the system. We conducted an analysis on the MEVA dataset and noted that very few intervals contained more than one ground-truth activity instance and fleetingly few contained more than two. This motivated employing an additional heuristic that limited the output to only contain the highest scoring detection for each class for each 3 s interval. We ran our system with this heuristic, resulting in an nAUDC and P_{miss} of 0.43871 and 0.61407, respectively, and without, resulting in an nAUDC and P_{miss} of 0.45612 and 0.62427, respectively. These results show that the heuristic yielded more accurate results, and, as such, we incorporated it into our system.

9.9 Results

Table 9.3 compares our results with the leaderboard for the 2020 ActEV challenge. The ActEV challenge requires that systems operate in real time and penalizes systems that operate slower than real time by computing time-limited nAUDC and P_{miss} values based on detections produced before the elapsed time limit. Table 9.3 reports the time-limited

Table 9.3. Our results compared to the 2020 ActEV challenge leaderboard as of July 19, 2020, ranked by penalized nAUDC. Starred entries report penalized nAUDC and P_{miss} values based on detections produced prior to the expiration of the real-time time limit.

Rank	System	Relative Processing Time	nAUDC	P_{miss}
1	Ours	0.164	0.35903	0.46970
2	UCF	0.722	0.36126	0.42337
3	CMU	0.498	0.38699	0.46352
4	UMD*	1.486	0.49867	0.53357
5	vireoJD-MM	0.149	0.53876	0.67389
6	UMCMU*	2.779	0.64138	0.68352
7	BUPT-MCPRL	0.969	0.61532	0.63203
8	CIS-JHU*	4.520	0.77784	0.79736
9	VUS*	1.344	1.00000	1.00000

results for the four starred entries that exceed the time limit. The table has been reranked accordingly. Our system outperforms the other systems in terms of nAUDC. It runs 4.4$\times$ and 3$\times$ faster than the UCF and CMU systems, respectively, which have comparable nAUDC scores. The only system to run faster than our system, vireoJD-MM, has considerably worse nAUDC and P_{miss} scores.

10. SUMMARY

10.1 Scorch

In this thesis research, I have presented a new framework for differentiable programming called Scorch. Scorch has native support for divide-and-conquer checkpointing and tensor streaming and is capable of using the GPU for computation. It can perform pointwise operations on tensors as well as most of the commonly used deep-learning primitives. Divide-and-conquer checkpointing coupled with tensor streaming lifts the memory bound on differentiable programs and allows users to train orders-of-magnitude deeper models or run extremely long-running gradient computations in differentiable programs, the likes of which are not possible using any existing deep-learning or differentiable-programming framework. We have presented several examples to show that Scorch can indeed seriously push the bounds of deep learning and differentiable programming using the GPU. We have run gradient descent on ResNet with 250,000 layers, DRANet with 250,000 layers and GPT-3 on a single GPU, something that no other prior work can do. I summarize the research contributions of this thesis document below.

10.2 Checkpointing

All known classical AD implementations that provide checkpointing only provide sublinear space penalty in special cases, like individual non-nested manually-annotated Fortran `DO` loops [47] and time-integration processes [72]. All known recent implementations of checkpointing in neural-network frameworks, including PyTorch, Superneurons, l2l, the two third-party enhancements to TensorFlow, and the method of Vieira [73], do not provide guaranteed sublinear space penalty and only apply to restricted forms of neural networks. Considerable prior work on training large neural networks attempts to migrate the tape to the CPU instead of performing a suitable form of checkpointing with sublinear space penalty. But this only makes it possible to store a larger tape and delays the inevitable memory exhaustion from the GPU to the CPU, allowing networks that are larger only by a constant factor; it does not yield sublinear space penalty for storing the tape. checkpointVLAD is the only known prior implementation of checkpointing to provide guaranteed sublinear space penalty and do so for arbitrary differentiable programs. However, it did not support tensors or GPU computation. SCORCH is the first and only implementation of checkpointing that provides guaranteed sublinear space penalty for arbitrary differentiable programs that use tensors or perform GPU computation.

10.3 Tensor Migration

The central contribution of our approach to tensor migration is its integration with divide-and-conquer checkpointing. Since divide-and-conquer checkpointing removes the bottleneck of GPU memory usage due to the tape, unlike all prior approaches to tensor migration in neural networks, we only stream the weights, not the tape. The weights have significantly different characteristics from the tape: they are more than an order of magnitude smaller and are read-only. The latter means that migration need only be unidirectional. Crucially, the former means that the time needed to migrate is less than the time needed to perform computation on the migrated values. This alleviates the need to solve an optimization problem to schedule the migration; simple profile-based scheduling suffices to achieve zero-overhead prefetching without stalling the computation. No other system makes the precise set of design tradeoffs to achieve this.

10.4 Object Classification from EEG Data

We have collected and released the largest single subject EEG dataset for visual object classification for a 1 out of 40 task. The dataset consists of 40,000 samples and is far from unsolved using the current state of the art methods. We have also released the analysis of using current state of the art methods on this dataset.

10.5 Spatio-temporal activity localization

We have shown a novel pipeline for tackling the problem of both spatially and temporally detecting, localizing and classifying an activity instance in an untrimmed, uncropped video, while being significantly faster than real time i.e. using very few compute resources. This makes this technique suitable for deployment on small scale devices with limited compute budget.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems*, San Diego, CA: NeurIPS, 2012, pp. 1097–1105.

[2] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Computer Vision and Pattern Recognition*, Piscataway, NJ: IEEE, 2016, pp. 770–778.

[3] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *International Conference on Learning Representations*, La Jolla, CA: ICLR, 2015.

[4] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *Computer Vision and Pattern Recognition*, Piscataway, NJ: IEEE, 2017, pp. 4700–4708.

[5] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *Computer Vision and Pattern Recognition*, Piscataway, NJ: IEEE, 2015, pp. 1–9.

[6] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the Inception architecture for computer vision,” in *Computer Vision and Pattern Recognition*, Piscataway, NJ: IEEE, 2016, pp. 2818–2826.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Stroudsburg, PA: ACL, 2019, pp. 4171–4186.

[8] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[9] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” *arXiv*, vol. 2005.14165, 2020.

[10] A. Griewank, “Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation,” *Optimization Methods & Software*, vol. 1, no. 1, pp. 35–54, 1992.

[11] R. Collobert, S. Bengio, and J. Mariéthoz, “Torch: A modular machine learning software library,” IDIAP, Research Report 02-46, 2002.

[12] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” in *International Conference on Multimedia*, New York, NY: ACM, 2014, pp. 675–678.

[13] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, “MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems,” *arXiv*, vol. 1512.01274, 2015.

[14] S. Tokui, K. Oono, S. Hido, and J. Clayton, “Chainer: A next-generation open source framework for deep learning,” in *NIPS Workshop on machine learning systems*, San Diego, CA: NeurIPS, 2015, pp. 1–6.

[15] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous distributed systems,” *arXiv*, vol. 1603.04467, 2016.

[16] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J. B. Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P.-L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M.-A. Côté, M. Côté, A. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. E. Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. Goodfellow, M. Graham, C. Gulcehre, P. Hamel, I. Harlouchet, J.-P. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrancois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P.-A. Manzagol, O. Mastropietro, R. T. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I. V. Serban, D. Serdyuk, S. Shabanian, É. Simon, S. Spieckermann, S. R. Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D. J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang, “Theano: A Python framework for fast computation of mathematical expressions,” *arXiv*, vol. 1605.02688, 2016.

[17] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Computer Vision and Pattern Recognition*, Piscataway, NJ: IEEE, 2016, pp. 779–788.

[18] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, San Diego, CA: NeurIPS, 2019, pp. 8026–8037.

[19] C. Mak and L. Ong, "A differential-form pullback programming language for higher-order reverse-mode automatic differentiation," in *POPL Workshop on Languages for Inference 2020*, New York, NY: ACM, 2020.

[20] A. Brunel, D. Mazza, and M. Pagani, "Backpropagation in the simply typed lambda-calculus with linear negation," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, 2019.

[21] M. Abadi and G. D. Plotkin, "A simple differentiable programming language," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, 2019.

[22] F. Wang, D. Zheng, J. Decker, X. Wu, G. M. Essertel, and T. Rompf, "Demystifying differentiable programming: Shift/reset the penultimate backpropagator," *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, 2018.

[23] A. Shaikhha, A. Fitzgibbon, D. Vytiniotis, and S. P. Jones, "Efficient differentiable programming in a functional array-processing language," *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, 2019.

[24] M. Huot, S. Staton, and M. Vákár, "Correctness of automatic differentiation via diffeologies and categorical gluing," in *International Conference on Foundations of Software Science and Computation Structures*, 2020, pp. 319–338.

[25] C. Elliott, "The simple essence of automatic differentiation," *Proceedings of the ACM on Programming Languages*, vol. 2, no. ICFP, 2018.

[26] D. Vytiniotis, D. Belov, R. Wei, G. Plotkin, and M. Abadi, "The differentiable curry," in *NIPS Workshop on Program Transformations*, San Diego, CA: NeurIPS, 2019.

[27] M. J. Innes, "Sense & sensitivities: The path to general-purpose algorithmic differentiation," in *Proceedings of Machine Learning and Systems*, 2020, pp. 58–69.

[28] G. Cruttwell, J. Gallagher, and D. Pronk, "Categorical semantics of a simple differential programming language," in *International Applied Category Theory Conference*, Electronic Proceedings in Theoretical Computer Science 328, 2020.

[29] R. E. Wengert, "A simple automatic derivative evaluation program," *Communications of the ACM)*, vol. 7, no. 8, pp. 463–464, 1964.

[30] B. Speelpenning, "Compiling fast partial derivatives of functions given by algorithms," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

[31] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[32] Y. M. Volin and G. M. Ostrovskii, "Automatic computation of derivatives with the use of the multilevel differentiating technique — I: Algorithmic basis," *Computers and Mathematics with Applications*, vol. 11, pp. 1099–1114, 1985.

[33] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, "Memory-efficient backpropagation through time," *arXiv*, vol. 1606.03401, 2016.

[34] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv*, vol. 1604.06174, 2016.

[35] J. M. Siskind and B. A. Pearlmutter, "Divide-and-conquer checkpointing for arbitrary programs with no user annotation," *Optimization Methods & Software*, vol. 33, no. 03–04, pp. 1288–1330, 2018.

[36] C. T. Haynes and D. P. Friedman, "Engines build process abstractions," in *Symposium on LISP and Functional Programming*, New York, NY: ACM, 1984, pp. 18–24.

[37] C. T. Haynes and D. P. Friedman, "Abstracting timed preemption with engines," *Computer Languages*, vol. 12, no. 2, pp. 109–121, 1987.

[38] R. K. Dybvig and R. Hieb, "Engines from continuations," *Computer Languages*, vol. 14, no. 2, pp. 109–123, 1989.

[39] J. C. Reynolds, "Definitional interpreters for higher-order programming languages," in *Proceedings of the 25th ACM National Conference*, Reprinted in *Higher Order and Symbolic Computation*, 11(4):363–397, 1998, New York, NY: ACM, 1972.

[40] G. L. Steele Jr. and G. J. Sussman, "Lambda, the ultimate imperative," MIT Artificial Intelligence Laboratory, A. I. Memo 353, 1976.

[41] J. C. Reynolds, “The discoveries of continuations,” *Lisp and Symbolic Computation*, vol. 6, pp. 233–247, 1993.

[42] A. W. Appel, *Compiling with continuations*. Cambridge, UK: Cambridge University Press, 2006.

[43] J. Kiefer and J. Wolfowitz, “Stochastic estimation of the maximum of a regression function,” *The Annals of Mathematical Statistics*, pp. 462–466, 1952.

[44] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization.,” *Journal of machine learning research*, vol. 12, no. 7, 2011.

[45] G. Hinton, N. Srivastava, and K. Swersky, “Neural networks for machine learning lecture 6a overview of mini-batch gradient descent,” *Coursera*, vol. 14, no. 8, p. 2, 2012.

[46] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *International Conference on Learning Representations*, La Jolla, CA: ICLR, 2015.

[47] L. Hascoët and V. Pascual, “TAPENADE 2.1 user’s guide,” INRIA, Rapport technique 300, 2004.

[48] W. Clinger and J. Rees, *Revised^4 report on the algorithmic language SCHEME*, Available as MIT Artificial Intelligence Laboratory Memo 848b, Nov. 1991.

[49] B. A. Pearlmutter and J. M. Siskind, “Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator,” *Transactions on Programming Languages and Systems*, vol. 30, no. 2, pp. 1–36, 2008.

[50] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, “cuDNN: Efficient primitives for deep learning,” *arXiv*, vol. 1410.0759, 2014.

[51] P. Vingelmann and F. H. Fitzek, *Cuda, release: 10.2.89*, 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit.

[52] H.-J. Boehm, “Space efficient conservative garbage collection,” *ACM SIGPLAN Notices*, vol. 28, no. 6, pp. 197–206, 1993.

[53] R. L. Graham, T. S. Woodall, and J. M. Squyres, “Open mpi: A flexible high performance mpi,” in *Parallel Processing and Applied Mathematics*, Berlin, Heidelberg: Springer, 2006, pp. 228–239.

[54] S. Jeaugey, "Nccl 2.0," in *GPU Technology Conference (GTC)*, 2017.

[55] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[56] F. Massa, S. Gross, S. Zagoruyko, S. Chintala, L. Yeager, eellison, E. Ranjan, A. A. Soltani, C. Pao, vfdev-5, G. Ren, R. Wightman, Y. Wu, R. Hataya, L. Roeder, P. Meier, F.-G. Fernandez, E. Gaasedelen, C. Beckham, bddppq, apache2046, A. Tejani, A. Mora-Fallas, driazati, B. Zhang, raghuramank100, and M. Kondela, *PYTORCH TORCHVISION implementation of ResNet*. [Online]. Available: https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py.

[57] R. Stojnic, R. Taylor, M. Kardas, V. Kerkez, and L. Viaud, *Papers with code*. [Online]. Available: https://paperswithcode.com/sota/semantic-segmentation-on-coco-stuff-test.

[58] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Europen Conference on Computer Vision*, Heidelberg, Germay: Springer, 2014, pp. 740–755.

[59] J. Fu, J. Liu, J. Jiang, Y. Li, Y. Bao, and H. Lu, "Scene segmentation with dual relation-aware attention network," *Transactions on Neural Networks and Learning Systems*, 2020.

[60] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," in *Computer Vision and Pattern Recognition*, Piscataway, NJ: IEEE, 2016, pp. 3213–3223.

[61] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, San Diego, CA: NeurIPS, 2017, pp. 5998–6008.

[62] OpenAI, *Openai TENSORFLOW implementation of gpt-2*. [Online]. Available: https://github.com/openai/gpt-2.

[63] N. Applied Deep Learning Research team, *Nvidia PYTORCH implementation of gpt-2*. [Online]. Available: https://github.com/NVIDIA/Megatron-LM.

[64] NVIDIA. (2020). "Openai presents gpt-3, a 175 billion parameters language model," [Online]. Available: https://news.developer.nvidia.com/openai-presents-gpt-3-a-175-billion-parameters-language-model/.

[65] J. Langston. (2020). "Microsoft announces new supercomputer, lays out vision for future ai work," [Online]. Available: https://blogs.microsoft.com/ai/openai-azure-supercomputer/.

[66] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Empirical Methods in Natural Language Processing*, Stroudsburg, PA: ACL, 2015, pp. 1412–1421.

[67] S. R. Bulo, L. Porzi, and P. Kontschieder, "In-place activated batchnorm for memory-optimized training of DNNs," in *Computer Vision and Pattern Recognition*, Piscataway, NJ: IEEE, 2018, pp. 5639–5647.

[68] B. Pudipeddi, M. Mesmakhosroshahi, J. Xi, and S. Bharadwaj, "Training large neural networks with constant memory using a new execution algorithm," *arXiv*, vol. 2002.05645, 2020.

[69] J. Feng and D. Huang, "Cutting down training memory by re-fowarding," 2018.

[70] N. Kukreja, J. Hückelheim, and G. J. Gorman, "Backpropagation for long sequences: Beyond memory constraints with constant overheads," *arXiv*, vol. 1806.01117, 2018.

[71] A. Griewank, D. Juedes, and J. Utke, "A package for the automatic differentiation of algorithms written in C/C++. User manual," Institute of Scientific Computing, Technical University of Dresden, Tech. Rep., 1996.

[72] A. Kowarz and A. Walther, "Optimal checkpointing for time-stepping procedures in ADOL-C," in *Computational Science*, ser. Lecture Notes in Computer Science, vol. 3994, Heidelberg, Germay: Springer, 2006, pp. 566–573.

[73] T. Vieira, *Memory efficient backpropagation thru time in a recurrent neural network*, 2016. [Online]. Available: https://gist.github.com/timvieira/b626e43496b87f84ec60e480cc9

[74] *Torch.utils.checkpoint*. [Online]. Available: https://pytorch.org/docs/stable/checkpoint.html.

[75] P. Goyal, *Trading compute for memory in pytorch models using checkpointing*. [Online]. Available: https://github.com/prigoyal/pytorch_memonger/blob/master/tutorial/Checkpointing_for_PyTorch_models.ipynb.

[76] T. Wang, P. Goyal, definitelynotmcarilli, H. Lee, B. Wignall, vfdev-5, T. Viehmann, X. Gao, pritamdamania, N. Shulga, mcarilli, soulitzer, Y. Jia, and C. Han. [Online]. Available: https://github.com/pytorch/pytorch/blob/master/torch/utils/checkpoint.py.

[77] C. AI, *Saving memory using gradient-checkpointing.* [Online]. Available: https://github.com/cybertronai/gradient-checkpointing.

[78] davisyoshida, *Tensorflow 2 gradient checkpointing.* [Online]. Available: https://github.com/davisyoshida/tf2-gradient-checkpointing.

[79] J. Zhang, S. H. Yeung, Y. Shu, B. He, and W. Wang, "Efficient memory management for GPU-based deep learning systems," *arXiv*, vol. 1903.06631, 2019.

[80] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "Superneurons: Dynamic GPU memory management for training deep neural networks," in *Principles and Practice of Parallel Programming*, New York, NY: ACM, 2018, pp. 41–53.

[81] M. Hildebrand, J. Khan, S. Trika, J. Lowe-Power, and V. Akella, "AutoTM: Automatic tensor movement in heterogeneous memory systems using integer linear programming," in *Architectural Support for Programming Languages and Operating Systems*, New York, NY: ACM, 2020, pp. 875–890.

[82] C.-C. Huang, G. Jin, and J. Li, "SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping," in *Architectural Support for Programming Languages and Operating Systems*, New York, NY: ACM, 2020, pp. 1341–1355.

[83] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *International Symposium on Microarchitecture*, Piscataway, NJ: IEEE, 2016, pp. 1–13.

[84] C. Meng, M. Sun, J. Yang, M. Qiu, and Y. Gu, "Training deeper models by GPU memory optimization on TensorFlow," in *NIPS ML Systems Workshop*, vol. 7, San Diego, CA: NeurIPS, 2017.

[85] H. Jin, B. Liu, W. Jiang, Y. Ma, X. Shi, B. He, and S. Zhao, "Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures," *Transactions on Architecture and Code Optimization*, vol. 15, no. 3, pp. 1–26, 2018.

[86] X. Chen, D. Z. Chen, and X. S. Hu, "moDNN: Memory optimal DNN training on GPUs," in *Design, Automation & Test in Europe Conference & Exhibition*, Piscataway, NJ: IEEE, 2018, pp. 13–18.

[87] T. D. Le, H. Imai, Y. Negishi, and K. Kawachiya, "Automatic GPU memory management for large neural models in TensorFlow," in *International Symposium on Memory Management*, New York, NY: ACM, 2019, pp. 1–13.

[88] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, “Capuchin: Tensor-based GPU memory management for deep learning,” in *Architectural Support for Programming Languages and Operating Systems*, New York, NY: ACM, 2020, pp. 891–905.

[89] J. Ren, J. Luo, K. Wu, M. Zhang, and D. Li, “Sentinel: Runtime data management on heterogeneous main memory systems for deep learning,” *arXiv*, vol. 1909.05182, 2019.

[90] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, “ZeRO-Offload: Democratizing billion-scale model training,” *arXiv*, vol. 2101.06840, 2021.

[91] J. Guo, W. Liu, W. Wang, C. Yao, J. Han, R. Li, Y. Lu, and S. Hu, “AccUDNN: A GPU memory efficient accelerator for training ultra-deep neural networks,” in *International Conference on Computer Design*, Piscataway, NJ: IEEE, 2019, pp. 65–72.

[92] C.-F. Jhu, P. Liu, and J.-J. Wu, “Data pinning and back propagation memory optimization for deep learning on GPU,” in *International Symposium on Computing and Networking*, 2018, pp. 19–28.

[93] T. Sekiyama, T. Imamichi, H. Imai, and R. Raymond, “Profile-guided memory optimization for deep neural networks,” *arXiv*, vol. 1804.10001, 2018.

[94] K. Shirahata, Y. Tomita, and A. Ike, “Memory reduction method for deep neural network training,” in *International Workshop on Machine Learning for Signal Processing*, Piscataway, NJ: IEEE, 2016, pp. 1–6.

[95] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, “Automatic CPU-GPU communication management and optimization,” in *Programming Language Design and Implementation*, New York, NY: ACM, 2011, pp. 142–151.

[96] R. Tezikov, *PyTorch implementation of L2L execution algorithm*, 2020. [Online]. Available: https://github.com/TezRomacH/layer-to-layer-pytorch.

[97] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, “Gist: Efficient data encoding for deep neural network training,” in *International Symposium on Computer Architecture (ISCA)*, New York, NY: ACM, 2018, pp. 776–789.

[98] A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse, “The reversible residual network: Backpropagation without storing activations,” *arXiv*, vol. 1707.04585, 2017.

[99] S. Kylasa, F. Roosta, M. W. Mahoney, and A. Grama, "Gpu accelerated sub-sampled newton's method for convex classification problems," in *Proceedings of the 2019 SIAM International Conference on Data Mining*, SIAM, 2019, pp. 702–710.

[100] A. S. Berahas, M. Jahani, P. Richtárik, and M. Takáč, "Quasi-newton methods for deep learning: Forget the past, just sample," 2020.

[101] B. Yuan, C. R. Wolfe, C. Dun, Y. Tang, A. Kyrillidis, and C. M. Jermaine, "Distributed learning of deep neural networks using independent subnet training," *arXiv preprint arXiv:1910.02120*, 2019.

[102] R. Li, J. S. Johansen, H. Ahmed, T. V. Ilyevsky, R. B. Wilbur, H. M. Bharadwaj, and J. M. Siskind, "The perils and pitfalls of block design for eeg classification experiments," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 1, pp. 316–333, 2020.

[103] C. Spampinato, S. Palazzo, I. Kavasidis, D. Giordano, N. Souly, and M. Shah, "Deep learning human mind for automated visual classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 6809–6817.

[104] V. J. Lawhern, A. J. Solon, N. R. Waytowich, S. M. Gordon, C. P. Hung, and B. J. Lance, "Eegnet: A compact convolutional neural network for eeg-based brain–computer interfaces," *Journal of neural engineering*, vol. 15, no. 5, p. 056 013, 2018.

[105] Y. Li, M. Murias, S. Major, G. Dawson, K. Dzirasa, L. Carin, and D. E. Carlson, "Targeting eeg/lfp synchrony with neural nets.," in *NIPS*, 2017, pp. 4620–4630.

[106] S. Palazzo, C. Spampinato, I. Kavasidis, D. Giordano, J. Schmidt, and M. Shah, "Decoding brain representations by multimodal learning of neural activity and visual features," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[107] W. Kay, J. Carreira, K. Simonyan, B. Zhang, C. Hillier, S. Vijayanarasimhan, F. Viola, T. Green, T. Back, P. Natsev, *et al.*, "The kinetics human action video dataset," *arXiv preprint arXiv:1705.06950*, 2017.

[108] K. Soomro, A. R. Zamir, and M. Shah, "Ucf101: A dataset of 101 human actions classes from videos in the wild," *arXiv preprint arXiv:1212.0402*, 2012.

[109] A. Piergiovanni and M. Ryoo, "Temporal gaussian mixture layer for videos," in *International Conference on Machine learning*, PMLR, 2019, pp. 5152–5161.

[110] S. Yeung, O. Russakovsky, N. Jin, M. Andriluka, G. Mori, and L. Fei-Fei, "Every moment counts: Dense detailed labeling of actions in complex videos," *International Journal of Computer Vision*, vol. 126, no. 2, pp. 375–389, 2018.

[111] H. Idrees, A. R. Zamir, Y. Jiang, A. Gorban, I. Laptev, R. Sukthankar, and M. Shah, "The thumos challenge on action recognition for videos "in the wild"," *Computer Vision and Image Understanding*, vol. 155, pp. 1–23, 2017.

[112] J. Lin, Y. Ding, W. Price, S. Han, and M. Li, *TSM: Temporal shift module for efficient video understanding*, Retrieved July 24, 2019 from `https://github.com/mit-han-lab/temporal-shift-module`, 2019.

VITA

Hamad Ahmed received his B.S. in Electrical Engineering from University of Engineering & Technology, Lahore, Pakistan in 2015. He is currently pursuing a Ph.D. in Electrical and Computer Engineering at Purdue University. His research primarily focuses on artificial intelligence, deep learning and computer vision.